VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky
a informatiky
Katedra informatiky

Programová podpora výuky UTI (zaměřená na konečné automaty)

Program support for education process in TCS (specialized to finite machines)

2009 **Bc. Vojtěch Gruchala**

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6.5.2009

.....

podpis diplomanta

Poděkování

Na tomto místě bych rád poděkoval vedoucímu diplomové práce Ing. Ondřeji Kohutovi za trpělivost, vstřícnost a odborné vedení této diplomové práce. Také své rodině a nejbližším přátelům za projevenou podporu.

Abstrakt

Cílem této diplomové práce bylo vytvořit plnohodnotné výukové softwarové dílo pro předmět teoretická informatika. Aplikace navazuje na mou bakalářskou práci z roku 2006, ale přesunula se z platformy kapesních počítačů na operační systémy Microsoft Windows, které jsou součástí většiny osobních počítačů. Celý vývoj systému byl řízen moderními nástroji, konkrétně softwarovým procesem RUP. Jednotlivé fáze procesu využívají digramy jazyka UML. Samotný program vysvětluje a podrobně popisuje velkou část problematiky konečných automatů. Operace jako převod automatu do normovaného tvaru, minimalizace, sjednocení či průnik dvou automatů i převod NDKA na DKA je možné nasimulovat pro téměř libovolný vstup definovaný uživatelem. Každým tímto algoritmem lze procházet vpřed i vzad. Největší důraz byl přitom kladen hlavně na správnost, názornost a znovupoužitelnost.

Klíčová slova

Teoretická informatika, konečný automat, deterministický, nedeterministický, normovaný tvar, minimalizace, sjednocení, průnik, převod, KA, DKA, ZNKA

Abstract

The aim of this thesis was to create an undepreciated tutorial software product for Theoretical Computer Science. The application continues where my bachelor´s work from 2006 ended. It was redesigned for operating system Microsoft Windows and personal computers from handheld platform. Whole system workflow was managed by modern utilities such as software process RUP. Every phase of the process uses UML diagrams. The program itself demonstrates and fully describes an extensive part of theory of finite state machines. The operations like normalization, minimalization, union or intersection of two machines and NFSM to FSM conversion are possible to simulate any input defined by user. Every algorithm can be browsed both forwards and backwards. The biggest emphasis was placed on precision, clearness and reusability.

Key words

Theoretical Computer Science, finite state machine, deterministic, non-deterministic, normalization, minimalization, union, intersection, conversion, FSM

Seznam použitých symbolů a zkratek

BP	– Bakalářská práce
BPM	– Business Process Modeling
DKA	– Deterministický Konečný Automat
DP	– Diplomová práce
FSM	– Finite State Machine (konečný automat)
KA	– Konečný Automat
MS	– Microsoft
NDKA	– Nedeterministický Konečný Automat
NFSM	– Non-Deterministic Finite State Machine
NKA	– Nedeterministický Konečný Automat
OOP	– Objektově Orientované Programování
OS	– Operační Systém
PC	– Personal Computer
PDA	– Personal Digital Asistent
RUP	– Rational Unified Process
SP	– Softwarový Proces
SWI	– Softwarové inženýrství
UML	– Unified Modeling Language
UTI	– Úvod do Teoretické Informatiky
XML	– eXtensible Markup Language
ZNKA	– Zobecněný Nedeterministický Konečný Automat

Obsah

1 Úvod 7
2 Historie aplikace 9
2.1 Bakalářská práce Petra Vašíčka 9
2.2 Bakalářská práce z roku 2006 11
3 Úvod do teoretické informatiky 14
3.1 Základní pojmy 14
3.2 Konečné automaty 15
3.3 Normovaný tvar 17
3.4 Minimalizace deterministického konečného automatu 17
3.4.1 Algoritmus minimalizace 18
3.5 Sjednocení a průnik dvou konečných automatů 20
3.6 Nedeterministický konečný automat 22
3.7 Převod nedeterministického automatu na deterministický 22
4 Softwarový proces projektu Automat 25
4.1 Základní typy softwarového procesu 25
4.2 RUP 26
4.3 Specifikace požadavků 27
4.3.1 Dokument Stakeholder Requests 27
4.3.2 Diagramy případů užití 29
4.4 Analýza a návrh 34
4.4.1 Aktivitní diagramy 35
4.5 Implementace 38
4.5.1 Třídní diagram 38
4.5.2 Struktura balíčků 40
4.6 Testování a předání 43
5 Třída Operace 45
5.1 Normování 45
5.2 Minimalizace 46
5.3 Převod NDKA na DKA 46
5.4 Sjednocení a průnik 47
6 Závěr 49
Použitá literatura 50
Příloha I – Obsah CD 51
Příloha II – Uživatelská dokumentace 52

1 Úvod

V dnešním světě se až příliš klade důraz na peníze, osobní kariéru a společenské postavení. Jen málokdo si ale přitom uvědomí, že toto všechno vychází ze vzdělání. Mnoho informací a poznatků získá člověk z praxe nebo jej to naučí život samotný. Přesto nelze popírat, že většinu vědomostí pobere už na základní, respektive střední škole, v lepším případě potom na půdě akademické.

Vzdělání je v dnešním světě jednou z nejdůležitějších hodnot. A hlavně jsou to nejlépe uložené peníze, čas i úsilí. To věděl již mnohem dříve známý americký státník, přírodovědec a spisovatel Benjamin Franklin, když řekl:

„Investice do vědění nesou nejvyšší úrok.“

Citát lze aplikovat na státní organizace úplně stejně, jako na každého jedince a moudro platí dnes možná ještě více, než když bylo před půl stoletím vyřčeno. A nelze s výrokem nesouhlasit, obzvláště pak v současné době, která vůbec není růžová. Člověk může ze dne na den ztratit práci, stejně jako přijít o své celoživotní úspory a majetek. Ovšem to, co Vám už nikdy nikdo nevezme, jsou Vaše vědomosti, zkušenosti a vzdělání. A právě proto jsem si za svou diplomovou práci zvolil aplikaci, která pomůže studentům vstřebat a překonat ne zrovna lehkou tematiku, se kterou se během studia nevyhnutelně setkají.

Tato diplomová práce je postavena na velice solidních základech. Prvotní vývoj započal již v roce 2004, kdy si podobné, ikdyž velice zjednodušené, téma vybral pro svou bakalářskou práci Petr Vašíček. Bez problémů ji obhájil, a aby jeho práce pouze nezapadla do knihovních archívů, rozhodl jsem se ji o rok později převzít a navázat na ni bakalářskou prací svou. Ta byla rovněž úspěšná na výbornou, protože rozšiřovala nejen funkčnost, ale také vnitřní strukturu a robustnost, protože jádro aplikace bylo přepsáno podle standardů objektového programování. O tom všem podrobně vypráví první kapitola následujícího textu.

Obě bakalářské práce však měly jednu velkou nevýhodu – byly napsány pouze pro kapesní počítače. To bránilo masivnějšímu rozšíření mezi studenty a částečně tím tak ztrácely smysl. Proto jsem se nyní rozhodl napsat a vytvořit plnohodnotnou výukovou aplikaci pro osobní počítače s operačním systémem Microsoft Windows. Přechod na pohodlnější ovládání a mnohem více prostoru na obrazovce se tak zákonitě musely projevit a na aplikaci je znát razantní posun k lepšímu. Uživatel má tak k dispozici program, který se příjemně a intuitivně ovládá, který bourá omezení malého displeje i kompatibility a hlavně přináší opravdu mnoho užitečných poznatků z problematiky teoretické informatiky a konečných automatů.

Základním úvodem teorie konečných automatů i teoretické informatiky nás provede další kapitola dále v textu. Budou objasněny nejdůležitější definice a pojmy, abychom mohli pochopit samotné fungování programu. Kromě čistě popisných definic a vět je vše doplněno o vysvětlení problematiky vlastními slovy, názorné obrázky, přechodové tabulky a jiné příklady.

Největší váhu celé diplomové práce má však program samotný. Už jeho návrhu bylo věnováno spoustu času a prostoru. Jak byl plánován, popisuje softwarový proces. Z velké části bylo využito modelu RUP, který je podrobně popsán v příslušné části textu. Krom něj se tam

také dočtete o všech fázích, které jsou doplněny odpovídajícími diagramy. Na těch lze nezasvěcené osobě nejlépe ukázat chování aplikace nebo jejích menších celků. Stejně tak diagramy popisují návrh, chování, vnitřní strukturu projektu i samotnou komunikaci a interakci mezi třídami či objekty. Většina fází obsahuje rovněž artefakty, které jsou přiloženy v elektronické podobě na CD.

V předposlední části textu jsem považoval za nutné popsat nejdůležitější části zdrojového kódu. Okomentovat si zaslouží každý hlavní samostatný formulář, kterých je v aplikaci celkem pět.Věnovat se budu rovněž některým vedlejším oknům, které jsou něčím specifické a od ostatních se odlišují. Budou to hlavně okna zobrazující postupy převodů, minimalizace nebo sjednocení a průniku. Tyto operace vyžadují složitější algoritmy, které jsou proto popsány obecným programovacím jazykem.

Žádnému textu nesmí chybět ani závěrečné shrnutí a zhodnocení výsledků celé diplomové práce. Nejinak tomu bude i v textu mém.

2 Historie aplikace

Původní nápad pro tuto diplomovou práci nevzešel z hlavy mé. Historie sahá trošku hlouběji, hned několik let zpátky. Kořeny sahají až do roku 2004, kdy bylo schváleno zadání bakalářské práce studentu Petru Vašíčkovi. Ten se rozhodl napsat výukový program k předmětu Základy teoretické informatiky pro mobilní zařízení, konkrétně typu kapesního počítače. Aplikace a s ní i celá práce nakonec nedopadly nejhůře, ale co je hlavní, nevyzněly naprázdno. Myšlenka totiž byla natolik dobrá (a provedení ne až tak špatné), že jsem hned další rok požádal o možnost navázat na práci svou bakalářkou. Svolení jsem dostal a původní funkcionalitu rozšířil a zdokonalil. I přesto se však stále jednalo o program s velice úzkým polem působnosti. PDA nebo jiný kapesní počítač vlastní jen malé procento studentů. Aby měla jakákoli výuková aplikace svůj smysl, musí se ke studentům dostat a šířit se mezi nimi dále. A jelikož téměř každý informatik na škole vlastní osobní počítač nebo notebook běžící na platformě Microsoft Windows, byla aplikace primárně pro tento systém jasnou volbou.

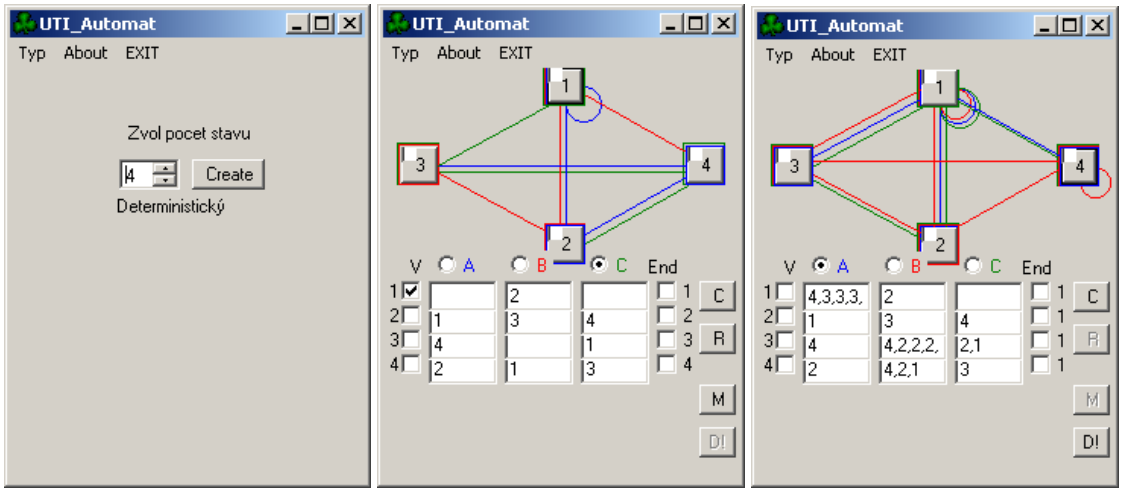
2.1 Bakalářská práce Petra Vašíčka

Samotná funkcionalita původní aplikace dosáhla postupem času zcela zásadnějších změn. Na začátku uměl program jen vytvořit a definovat oba typy automatů s pevným počtem písmen a dvěma až osmi stavy, tento automat pak bylo možno minimalizovat a v případě nedeterministického automatu také převést na deterministický, přičemž se algoritmus pouze uložil do souboru, na který byl uživatel upozorněn vyskočeným informačním oknem. Aplikace byla omezena možnostmi displeje kapesního počítače, proto ji bylo možno celou ovládat pouze myší, resp. stylusem. K záporným vlastnostem PDA však patřilo hlavně nízké rozlišení a málo prostoru na obrazovce, čemuž musel být program přizpůsoben. Bohužel měla práce ještě dva nedostatky, na které se přišlo až při vývoji navazující aplikace. Oba celkem zásadní – nefungoval správně ani převod nedeterministického automatu na deterministický, ani algoritmus minimalizace. Ony tedy byly správné, ale jen pro některé zadání, což rozhodně nebylo postačující. Dalším obrovským nedostatkem byla nestabilita. Program se zhroutil už po několika málo zadaných operacích, neočekávané vstupy pak nebyly ošetřeny téměř vůbec a tak jsme si museli zvyknout na častou ztrátu pečlivě zadávaných dat a následné restartování nebo znovuspuštění aplikace.

Navíc ani nebyl program nikterak uživatelsky přívětivý. Menu bylo velice strohé, prakticky pouze tři položky – změna typu automatu, stručné informace o autorovi a volba ukončení. Tedy nulová možnost nastavení a personalizace. Vše ještě ke všemu v kombinovaném česko-anglickém názvosloví (tlačítka *About*, *Exit* či *Create* proti všem ostatním českým popiskům). O načítání nebo ukládání automatů také nemohla být ani řeč. Tím se program stal velice jednoúčelným. Po jeho spuštění jste byli vyzváni ke zvolení typu automatu a počtu stavů. Ten byl omezen na maximálních osm. Počet písmen abecedy byl stanovený napevno a to na tři symboly. Po potvrzení volby již nevedla žádná cesta zpět, ani nebylo možno zadání jakkoli editovat.

Vše podstatné se dále odehrávalo na této druhé stránce. Uživatel musel nejprve automat definovat, zvolit počáteční a přijímající stavy, stejně jako zakreslit či zadat přechody. Vytváření samotného automatu však student zvládl na výbornou, především myšlenka naklikání přechodů či jejich překreslení do grafu z tabulky. Rovněž provedení stálo za to převzít a realizovat obdobně. Dynamické generování ovládacích prvků i řádků tabulky bylo zvoleno vhodně, na pozadí však byla jednoduchá myšlenka – rozmístit všechny tyto komponenty napevno dopředu a poté na samotné plátno aplikace zobrazit jen minimální nutný počet z nich. Lišácké, ale fungující i dobře vypadající.

Bohužel, další akce už byly řešeny méně šťastně. Funkci čtyř tlačítek s jednopísmenným popiskem musela poodhalit až uživatelská dokumentace. Písmeno *C* byla zkratka pro anglické slovo *Clear* (vymazání) a reprezentovalo smazání celé přechodové tabulky i grafu. Znak *R* nebyl kupodivu očekávaný *Reset*, ale *Repaint*, což v překladu znamená překreslení, v našem případě přechodové tabulky do grafu. Zbyly nám ještě tlačítka s popisky *M* a *D* s vykřičníkem. Písmenem *M* se spouštěla minimalizace zadaného vytvořeného automatu. Jak ale uživatel záhy zjistil, minimalizace po několika voláních (stiscích *M*) stále měnila svůj výsledek až do konečného stavu, kdy zůstal jeden jediný stav. To bylo samozřejmě špatně, protože minimální automat může být pouze jeden pro každý libovolně zadaný. Poslední tlačítko se znaky *D!* bylo někdy zašedlé, což napovídalo, že je aktivní pouze u nedeterministických automatů a slouží k jejich převodu na deterministické. Výsledek se přitom z převážné většiny uložil pouze do souboru a nebylo tak možno s výsledným automatem dále pracovat. Co bylo horší, tak ani formát dat v souboru nebyl lehce rozluštitelný a představit si pod tou směsicí písmen a čísel jakýkoli automat nebo dokonce postup převodu bylo těžké.



Obr. 1 – Ukázky obrazovek aplikace Petra Vašíčka

Zbývá rozebrat vnitřní strukturu projektu a zdrojový kód samotný. Začnu programovacím jazykem a obecně stylem psaní kódu. Student měl určitě slušnou znalost jazyka C#, ale rozhodně nebyl schopen efektivně využít jeho kompletních konstrukcí, možností stavby příkazů a hlavně členění do bloků. Také některé funkce byly volány zbytečně složitě. Metody a funkce měly mnohdy více než stovky řádku, čímž utrpěla celková přehlednost. Ke všemu úplně

chyběla jakákoli projektová dokumentace nebo alespoň komentáře integrované přímo ve zdrojových kódech. Co poznal dokonce i laik na první pohled, byla absence ošetření jakýchkoli výjimek. To bylo nezbytné, protože aplikace umožňovala uživateli zadat libovolný vstup z klávesnice a bylo tedy nutné předpokládat, že dojde k záměrnému nebo náhodnému odeslání neplatných dat. V kódu však nic ošetřeno nebylo. Proto při vložení vysokého čísla neplatného neexistujícího stavu, nebo dokonce písmena či jiných znaků namísto číslice, se snažil program vnitřně s těmito daty pracovat a v drtivé většině došlo k selhání a následnému pádu aplikace, v krajním případě až zamrznutí celého počítače.

Struktura projektu také neměla žádný řád. Třídy nebyly nikterak uspořádané, všechny dokonce v jednom adresáři, a chyběl jakýkoli náznak moderního postupu vývoje aplikace. Bylo vidět, že fáze analýzy, návrhu i plánování byly zcela přeskočeno a začalo se rovnou programovat. Nesetkáme se proto s žádnou dědičností ani polymorfismem. Po objektově orientovaném programování žádná stopa. Alespoň byly striktně odděleny jednotlivé formuláře, o což se ale stará samotné vývojové prostředí Visual Studio. Samostatnou jednotku ještě tvořila třída *Operace*. Ta měla původně asi oddělovat od grafických formulářů část logiky se sofistikovanějšími algoritmy, ve výsledku ale obsahovala pouze jednu metodu pro setřídění vstupních čísel a veškeré opravdu složité algoritmy, mezi které rozhodně patřily minimalizace a převod nedeterministického automatu na deterministický, tak stejně zůstaly zakomponovány mezi grafickými prvky oken.

2.2 Bakalářská práce z roku 2006

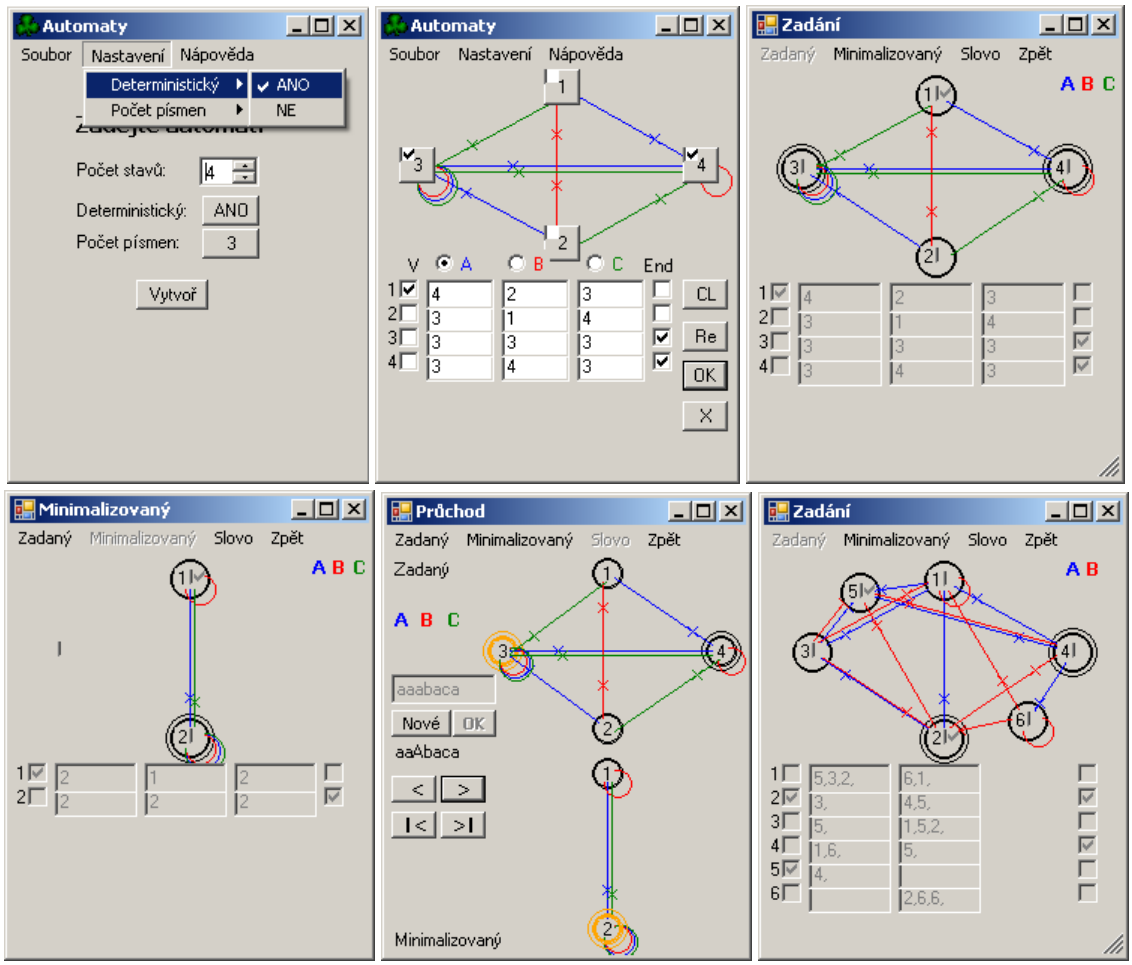
Při vypracovávání mé vlastní bakalářské práce jsem se především snažil poučit z chyb svého předchůdce. Než jsem začal psát samotný kód, zaměřil jsem se na důkladnou analýzu, vytipoval společné rysy objektů a rozpoznal redundanci v původních třídách. Rovněž jsem si rozvrhl logiku hned do několika formulářů. Při návrhu jsem pak využil poznatků ze studia a uplatnil znalost objektově orientovaného programování. Deterministické i nedeterministické automaty měly hned několik společných prvků a proměnných, ze kterých jsem vytvořil abstraktní mateřskou třídu. Z té jsem děděním vytvořil třídy *AutomatD* a *AutomatND*, které odpovídaly výše zmíněným typům. Navíc je každý automat tvořen stavy. U nedeterministického automatu platí ale jiná pravidla pro přechody, proto i zde vznikli z abstraktní třídy Stav dva konkrétní potomci StavD a StavND. Kromě těchto tříd reprezentujících objekty byly vytvořeny také dvě pomocné – *Metody.cs* a *Hlavni.cs*.

První jmenovaná měla podobný význam, jako *Operace* v předešlé práci. Jen tuto roli už plnila. Sloužila k ověřování vstupních řetězců, validity dat, platnosti rozsahu, ale i třídění a filtrování vstupních údajů. Jak už z názvu *Hlavni.cs* vyplývá, zcela zásadnější byla tato třída. Ve svém těle skrývala pod veřejnými hlavičkami názvů metod takové procedury, které se staraly o ty nejtěžší a nejvýkonnější části celého projektu. Mezi ostatními vyčnívaly hlavně minimalizace, převod NDKA na DKA, převod na normovaný tvar a také vyhodnocení slova pro průchod automaty. Samozřejmě i další metody měly svou váhu, některé z nich i klíčovou roli, ale podrobně se zde jimi zabývat nebudu.

Každá třída měla patřičně nastavenou viditelnost v rámci projektu. Atributy a metody byly pojmenovány podle obecných konvencí. Rozsáhlé procedury byly navíc rozděleny na menší celky, přičemž často se opakující sekvence kódu byly vyčleněny do funkcí separátních. Nechyběla důkladná kontrola kompatibility všech možných (a často i nemožných) datových typů. Tam, kde to nestačilo, bylo nasazeno pečlivé hierarchické ošetření výjimek, včetně zotavení aplikace po neplatných vstupech.

Jak můžete vidět na obrázku 2, velkých změn dostálo i uživatelské rozhraní. Jak již bylo zmíněno, přibyla další okna formulářů, což zvýšilo přehlednost a hlavně ještě více zjednodušilo ovládání.

Zcela zásadní proměnou prošlo menu. Pod položkou *Soubor* se skrývá jednak volba *Nový*, kterou zahájíte nebo znovu vyvoláte vytvoření nového automatu, tlačítko *Zavřít* se zřejmým efektem a také volba *Příklad*. Zde je uživateli nabídnuto načtení jednoho ze čtyř připravených předdefinovaných automatů, na kterých jsou všechny algoritmy názorně viditelné. V menu *Nastavení* lze při zadávání parametrů vytvářeného automatu nastavit to samé, co v okně úvodní obrazovky. A konečně *Nápověda* prozradí něco málo o *Ovládání*, *O autorovi* a *O programu* samotném.



Obr. 2 – Ukázky obrazovek mé bakalářské práce

Samotná obrazovka definování automatu se mnoha změn nedočkala. Byla přejmenována tlačítka, přibylo jedno pro návrat a pro potvrzení (ukončení) zadávání. Při ručním vkládání přechodu do tabulky již není povoleno zadat písmeno, stejně jako program odmítne jakoukoli neplatnou číslici. Při opakovaném zadání stejného přechodu u nedeterministického automatu dojde k odfiltrování všech duplicit a jiných neplatných znaků.

Po potvrzení se zobrazí nová obrazovka, kde je již automat přehledně překreslen i s přechodovou tabulkou. Počáteční stav je označen zatržítkem, přijímající potom dvojitou kružnicí, jak jsme zvyklí. Žádné údaje nejsou už v tuto chvíli editovatelné. Změnilo se rovněž menu, kterým lze nyní přepínat mezi zadáním, minimalizací a převodem nebo průchodem.

Minimalizovaný automat má v tomto smyslu dvě různé funkce. Pokud se jedná o automat nedeterministický, dojde před samotnou minimalizací ještě k převodu. Jinak se chová aplikace standardně dle očekávání. Postup převodu i samotné minimalizace se vypíše do souboru, což je uživateli pouze oznámeno pomocí informačního okna.

Poslední částí aplikace byl průchod automatem na základě zadaného slova. Jednalo se o velice užitečné rozšíření, na které bylo rozhodně nutné opět navázat v práci diplomové. Zde fungovalo jednoduše. Zobrazily se dva automaty – zadaný a minimalizovaný (nebo převedený). Počáteční stavy byly barevně zvýrazněny. Uživatel nyní mohl zadat libovolné slovo, ze kterého byly automaticky odstraněny neplatné symboly abecedy. Písmeny tohoto slova pak probíhal průchod oběma automaty současně, aby bylo možné pozorovat, v čem se oba liší.

Tato druhá bakalářská práce již byla mnohem zdařilejší. I zde se ale objevila chyba v algoritmu minimalizace a ten bylo nutné celý přepsat. Rovněž architektura nebyla natolik robustní, aby se na ní dala postavit plnohodnotná aplikace pro operační systém Windows. Základní prvky však bylo možné převzít a tak jsem měl do začátku tvorby diplomové práce k dispozici kvalitní stavební kameny.

3 Úvod do teoretické informatiky

Teoretická informatika je obor, které úzce souvisí s potřebami praxe při vývoji software, hardware, ale také obecně při návrhu a vytváření systémů. Mezi základní odvětví této vědy patří teorie jazyků a automatů. Hlavím cílem je pak matematicky popsat nejrůznější algoritmické výpočty, jejich proveditelnost a v neposlední řadě i složitost. Pro co nejpřesnější popis vstupů a očekávaných výstupů je ale nutné nejdříve definovat abecedu symbolů a formálních slov nad touto abecedou.

V této části textu se proto budu věnovat především vysvětlení základů problematiky, zavedu definici používaných pojmů, které se pokusím popsat i vlastními slovy, a vybrané algoritmy předvedu na příkladech.

3.1 Základní pojmy

Aby bylo zadávání a výpis výsledků jednotné, je potřeba najít a zavést společné značení. Řetězec znaků neboli symbolů předávané informace pak budeme nazývat jednoduše slovem. Proto uvedu následující definice a značení.

- *Abecedou* budeme mít na mysli libovolnou konečnou množinu $\sum$, jejíž prvky nazýváme symboly (písmena) abecedy. Například $\sum = \{a, b, c\}$.

- *Slovem* nad abecedou $\sum$ myslíme libovolnou konečnou posloupnost prvků $\sum$, např. 'a', 'c', 'bac', 'babacca'. Prázdné slovo (neobsahuje žádný symbol) je také slovem a značí se znakem epsilon – ε.

- *Délkou* slova chápeme počet znaků (písmen) ve slově. Prázdné slovo ε má délku nula.

- *Jazykem* nad abecedou $\sum$ pak myslíme libovolnou podmnožinu L slov abecedy $\sum^*$. Výrazem $\sum^*$ označujeme množinu všech slov nad abecedou $\sum$.

Mezi další důležité pojmy patří například *prefix* (předpona), *sufix* (přípona) a *zřetězení slov*, které označuje spojení dvou a více slov těsně za sebe bez mezer v pevně daném pořadí tak, že vznikne slovo nové.

S formálními jazyky lze rovněž provádět různé operace. Jedná se hlavně o operace klasické množinové (sjednocení, průnik a rozdíl), zřetězení jazyků dvou různých (obdobné zřetězení slov) nebo rekurentní zřetězení jazyka jednoho (tzv. iterace).

3.2 Konečné automaty

Neformálně je konečný automat (KA) popisující procesy s předem omezeným počtem možných stavů nejjednodušší klasický výpočetní model. Je to uzavřený systém, který se může nacházet v nekonečně mnoha vnitřních stavech. Přechody mezi těmito stavy jsou přitom předem jasně definovány a jsou vyvolány vnějším podnětem. Každý automat pak někde musí mít svůj začátek (nějakou výchozí podobu), který se nazývá počáteční stav. Navíc každý stav musí mít určeno, zdali je přijímající či nikoli.

Konkrétní jeden konečný automat se může zadávat jako ohodnocený orientovaný graf. Tento graf automatu je tvořen vrcholy, které reprezentují jednotlivé vnitřní stavy a ohodnocené orientované hrany (šipky) ukazují přechody mezi stavy pro jednotlivé vstupy. Stavy jsou nejčastěji očíslovány od jedničky až po počet stavů, přičemž první je většinou stav počáteční. Vstupní podněty jsou značeny symboly abecedy. Spojením vstupních symbolů pak vzniká slovo.

Toto neformální zadání má však také svou přesnou definici:

Konečný automat je uspořádaná pětice $A = (Q, \Sigma, \delta, q_0, F)$**,** kde

- Q je konečná neprázdná množina stavů,
- Σ je konečná neprázdná množina zvaná vstupní abeceda,
- $\delta : Q \times \Sigma \rightarrow Q$ je přechodová funkce,
- $q_0 \in Q$ je počáteční (iniciální) stav
- $F \subseteq Q$ je neprázdná množina přijímajících (koncových) stavů.

Přechodová funkce δ má dva argumenty $\delta\,(q; x)$. Pokud se automat právě nachází ve stavu q a čte ze vstupu znak x, musí proběhnout přechod do dalšího stavu $\delta\,(q; x)$ (ten může být jiný i stejný jako původní q). Hlavní ale je, že pro každý stav a každý vstupní symbol musí být jasně definováno, kam automat přejde.

Počáteční stav q_0 musí být pouze jeden, ale automat může mít více přijímajících stavů. Těm se sice často říká koncové, což ale vůbec neznamená, že by v nich výpočet vždy končil.

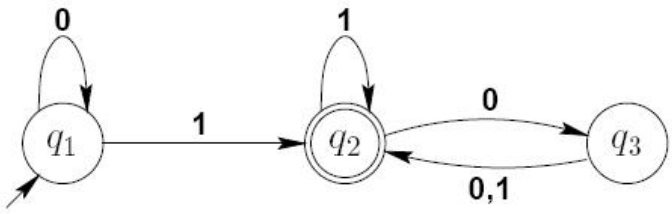
Grafem automatu (neboli stavovým diagramem) rozumíme orientovaný ohodnocený graf, ve kterém:

- vrcholy jsou stavy automatu, tj. množina Q,
- počáteční stav (q_0) je vyznačený šipkou směřující do stavu a koncové stavy (F) dvojitým kruhem,
- hrana z vrcholu u do vrcholu v je označena výčtem všech písmen abecedy, které stav u převádějí na stav v, tj. $\{ x \in \Sigma : \delta(u; x) = v \}$.

Kompletně zadaný automat může vypadat například takto:

- množina stavů $Q = \{q_1, q_2, q_3\}$,
- vstupní abeceda $\Sigma = \{0, 1\}$,
- přechodové funkce $\delta(q_1, 0) = q_1$, $\delta(q_1, 1) = q_2$, $\delta(q_2, 0) = q_3$, $\delta(q_2, 1) = q_2$, $\delta(q_3, 0) = q_2$, $\delta(q_3, 1) = q_2$,
- počáteční stav je q_1 a
- množina přijímacích stavů $F = \{ q_2 \}$.

Graf takového automatu je pro lepší představu znázorněn na obrázku 3.



Obr. 3 – graf automatu se třemi stavy q_1, q_2, q_3 nad abecedou {0, 1}

Pojďme se nyní na tento automat podívat blíže. Zkusíme si například otestovat slovo „1101“.
Z definice i grafu víme, že počáteční stav je q_1. Pokud chceme automatem procházet na základě zadaného slova, musíme po jednom projít jeho symboly. Prvním z nich je znak 1. Podle přechodové funkce se tímto znakem dostaneme do stavu q_2. Pokračujeme druhým symbolem, kterým je opět jednička. Změnila se ovšem přechodová funkce. Ta nám říká, že i po uskutečnění přechodu zůstaneme ve stavu q_2. Následuje nula, díky které se přesuneme do stavu q_3. Posledním symbolem je opět jednička, která nás vrátí do vrcholu q_2. Žádná další písmena ve slově nejsou, proto musí vyhodnotit, bylo-li slovo automatem přijato či nikoli. Stav q_2 se nachází v množině přijímajících stavů a proto je i celé slovo „1101“ přijato.

Po několika pokusech zjistíte, která slova jsou automatem přijímána (1, 111, 101, 100001, 10100) a která nikoli (0, 000, 10, 11010). Zkušenější čtenář už je možná schopen odvodit, že automat přijme všechna slova, která končí jedničkou, nebo obsahují alespoň jednu jedničku a dvě nuly na konci.

Aby však bylo možné automatem procházet přehledně i bez grafu, vznikla tzv. přechodová tabulka. Tvoří ji řádky označující stavy automatu a sloupce označené symboly abecedy. Políčko na řádku *q* a sloupci *a* udává přechodovou funkci $\delta\ (q, a)$. Počáteční stav je značený $\rightarrow$ a přijímající $\leftarrow$. Konkrétní strukturu si ukážeme na předchozím příkladu.

	'0'	'1'
$\rightarrow q_1$	q_1	q_2
$\leftarrow q_2$	q_3	q_2
q_3	q_2	q_2

Obecně lze průchod automatem na základě slova *s* (nebo též výpočet konečného automatu na vstupním slově *s*) popsat takto:

1. Začneme v počátečním stavu q_0 na začátku slova *s*.
2. Přečteme aktuální symbol x slova *s* a přejdeme do stavu určeného přechodem $\delta(q, x)$, kde q je současný stav automatu. Zároveň se posuneme na následující písmeno slova *s*.
3. Opakujeme bod 2, dokud nejsou přečtena všechna písmena slova *s*.
4. Pokud je poslední stav automatu přijímající, pak je také celé slovo *s* přijato, v opačném případě je zamítnuto.

3.3 Normovaný tvar

Ještě než uvedeme definici automatu v normovaném tvaru, bylo by vhodné objasnit pojem dosažitelnost, resp. nedosažitelnost stavu.

Stav q automatu A je **dosažitelný** slovem w, pokud se výpočet A po přečtení celého slova w zastaví ve stavu q.

V automatu tak mohou být stavy, do kterých nevede žádný sled z počátečního stavu q_0. Takové stavy jsou zcela jasně zbytečné a nazývají se **nedosažitelné**. Těmto stavům se snažíme při vytváření automatu vyhnout, a když už zadány byly, je vhodné je odstranit. Nyní už můžeme přejít k samotné definici normovaného tvaru:

Automat je v **normovaném tvaru** právě tehdy, když jsou jeho stavy očíslovány 1, 2, 3,… v abecedním pořadí nejmenších slov, kterými lze tyto stavy dosáhnout.

Když se nad definicí trošku zamyslíme, přímo z ní vyplývá, že dojde k odstranění nedosažitelných stavů. Nikde ale zatím nebylo řečeno, jak takový automat vytvoříme. Postup je přitom jednoduchý a nejvíce se podobá průchodu grafu do šířky, při kterém souběžně dochází k přečíslování vrcholů. Postup normování lze popsat těmito kroky:

1. Počáteční stav označíme číslem 1.
2. Dále, např. v případě abecedy {a, b}, zjistíme stav q, do něhož automat přejde ze stavu 1 symbolem a; když q není označen, označíme jej číslem 2.
3. Pak zjistíme stav q, do něhož automat přejde ze stavu 1 symbolem b; když q není dosud označen, označíme jej nejmenším dosud nepoužitým číslem.
4. Otestovali a prošli jsme přechody stavu 1, pokračujeme stavem 2 atd., dokud nezískáme všechny dosažitelné stavy.

3.4 Minimalizace deterministického konečného automatu

Často se stává, že existují dva na první pohled zcela odlišné automaty, které rozpoznávají stejný jazyk. Někdy mívají různý počet stavů, jindy zase jiné přechody nebo jsou stavy jen očíslované v nahodilém pořadí. Nabízí se tedy otázka, jak jednoznačně určit, jestli dva různé automaty jsou

ekvivalentní, tedy jestli rozpoznávají opravdu stejný jazyk. Přitom by se hodilo, aby byl výsledný automat nejmenší, co se počtu stavů týče. Dále je samozřejmé, aby byl v normovaném tvaru. Přesně takovýto automat lze z každého získat procesem minimalizace a nazýváme jej pak automatem minimálním (někdy také minimalizovaným). Tento jednoduchý matematický algoritmus minimalizace má svou jasně definovanou podobu a tu si nyní ukážeme.

Začal bych důležitou definicí, která říká:

Dva konečné automaty A_1 a A_2 nazveme (jazykově) **ekvivalentní**, jestliže přijímají týž jazyk, tj. jestliže $L(A_1) = L(A_2)$.

Na toto tvrzení navážeme přímo objasněním pojmu minimalizace, které vlastně shrnuje do definice to, co jsme si neformálně naznačili v úvodu:

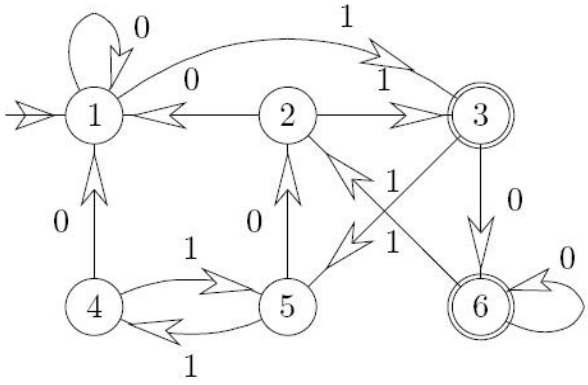
Konečný automat nazveme **minimálním automatem**, jestliže neexistuje automat, který by s ním byl ekvivalentní a měl by menší počet stavů.

3.4.1 Algoritmus minimalizace

Mějme daný automat $A = (Q, \Sigma, \delta, q_0, F)$ bez nedosažitelných stavů.

1. Začneme rozkladem $R_0 = \{Q \setminus F, F\}$ množiny všech stavů A na ty přijímající a nepřijímající.
2. Nechť v kroku $k \geq 0$ máme rozklad na $R_k = \{P_1, P_2, \ldots, Pm\}$ množiny všech stavů. Pro všechna $i \in \{1, 2, ..., m\}$ a $a \in \Sigma$ uděláme následující: Rozložíme P_i na rozklad P_i^A podle toho, do kterých množin z R_k vedou ze stavů v P_i šipky se symbolem a.
3. Uděláme sjednocení průniků těchto (mini-)rozkladů $R_{k+1} = \bigcup_i \bigcap_a P_i^A$. Tím získáme všechna možná další rozlišení uvnitř tříd rozkladu R_k všemi znaky abecedy.
4. Pokud $R_{k+1} \neq R_k$, tj. došlo k dalšímu rozdělení v rozkladu, vracíme se krokem k+1 na bod 2.
5. Jinak nechť R je množina stavů po jednom vybraných z tříd rozkladu R_k (reprezentanti, přitom q_0 je také vybráno) a δ' je restrikce přechodové funkce δ na jednotlivé třídy rozkladu R_k. Pak minimální automat je $A_0 = (R, \Sigma, \delta', q_0, F \cap R)$.

Abychom mohli tento složitě popsaný algoritmus snadno pochopit, ukážeme si jej na konkrétním příkladu pro automat na obrázku 4.



Obr. 4 – Deterministický konečný automat **před minimalizací**

Takto vypadá přechodová tabulka pro náš automat (vlevo), vpravo pak vidíte vyplněnou tabulku pro první rozklad R_0 = { {1, 2, 4, 5}, {3, 6}}.

	0	1
→ 1	1	3
2	1	3
← 3	6	5
4	1	5
5	2	4
← 6	6	2

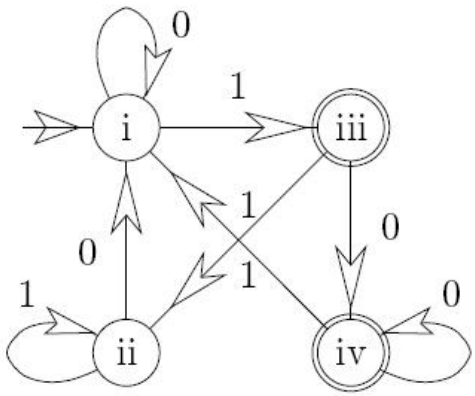
		0	1
i	1	i	ii
i	2	i	ii
i	4	i	i
i	5	i	i
ii	3	ii	i
ii	6	ii	i

První podmnožina rozkladu se tak dále rozpadá na množiny {1; 2} a {4; 5}, mezi kterými lze rozlišit přechodem při znaku 1. Pokračujeme obdobně dalšími kroky, čímž získáme další dva rozklady R_1 a R_2 s následujícími přechodovými tabulkami:

		0	1
i	1	i	iii
i	2	i	iii
ii	4	i	ii
ii	5	i	ii
iii	3	iii	ii
iii	6	iii	i

		0	1
i	1	i	iii
i	2	i	iii
ii	4	i	ii
ii	5	i	ii
iii	3	iv	ii
iv	6	iv	i

Poslední tabulka (vpravo) nám udává rozklad R_2 = {{1, 2}, {4, 5}, {3}, {6}} , který již žádným ze znaků 0 nebo 1 nelze více rozložit (zjemnit), a proto je automat (na obrázku 5) daný touto tabulkou minimální.



Obr. 5 – Deterministický konečný automat **po minimalizaci**

Pokud chceme rozhodnout o ekvivalenci libovolných dvou deterministických konečných automatů, stačí oba minimalizovat, převést do normovaného tvaru a porovnat jejich přechodové tabulky, resp. isomorfismus grafů.

3.5 Sjednocení a průnik dvou konečných automatů

Navrhnout automat, který rozpoznává jednoduchý regulární jazyk, není pro zkušenějšího člověka se základní znalostí výše popisované tematiky až takový problém. Pokud by ovšem měl sestavit automat rozpoznávající současně několik jazyků, tak už by to nejspíše v rozumném čase nedokázal. Naštěstí ale existuje způsob, který umožňuje sjednocení (a dokonce i průnik) dvou regulárních jazyků jednoduše automatizovat. Je vhodné si také uvědomit, že opětovným opakováním algoritmu můžeme spojit vlastně libovolný počet jazyků do jednoho.

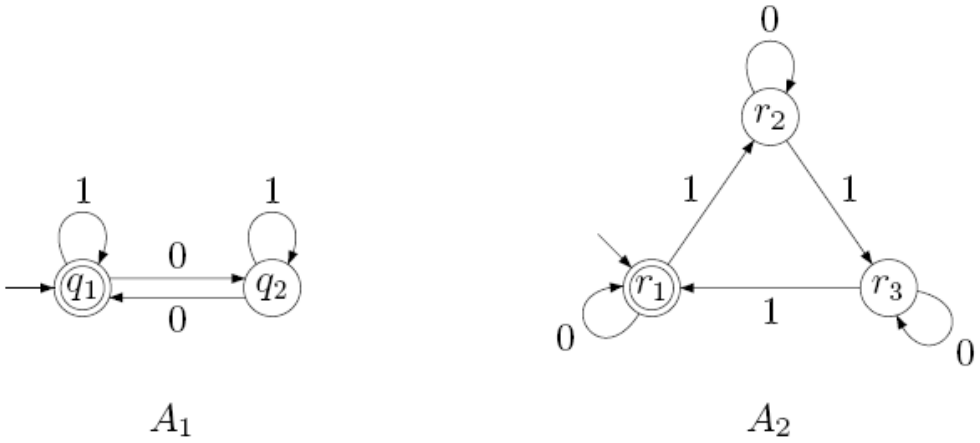
Obecný popis konstrukce výsledného automatu lze formálně zapsat takto:

Nechť L_1 (resp. L_2) je regulární jazyk rozpoznávaný automatem A_1 (resp. A_2), formálně zapsáno $L_1 = L(A_1)$, $L_2 = L(A_2)$.
Konečné automaty jsou přitom definovány jako $A_1 = (Q_1,\Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2,\Sigma, \delta_2, q_{02}, F_2)$.
Definujme automat $A = (Q,\Sigma, \delta, q_0, F)$ takto:

- $Q = Q_1 \times Q_2$,
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ pro všechna $q_1 \in Q_1$, $q_2 \in Q_2$, $a \in \Sigma$,
- $q_0 = (q_{01}, q_{02})$,
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.

Z této definice automatu zcela jasně plyne, že pro libovolné dva stavy $q_1 \in Q_1$, $q_2 \in Q_2$ a libovolné slovo w ($w \in \Sigma^*$) je přechod $\delta((q_1, q_2),w) = (\delta_1(q_1,w), \delta_2(q_2,w))$. Jinými slovy řečeno, každým vstupním slovem w automat A přejde do stavu (q_1, q_2), kde q_1 je stav automatu A_1 dosažený slovem w a obdobně q_2 je příslušný stav automatu A_2. Z toho snadno plyne, že výsledný regulární jazyk L rozpoznává stejná slova, jako sjednocení jazyků L_1 a L_2:
$L(A) = L_1 \cup L_2$.

Postup konstrukce automatu reprezentujícího sjednocení dvou jazyků se pokusím vysvětlit na následujícím příkladu pro automaty na obrázku 6.



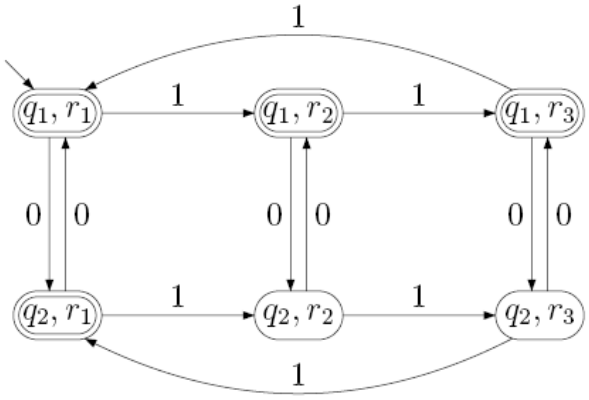
Obr. 6 – Dva jednoduché deterministické konečné automaty A_1 a A_2

Již z obrázku je patrné, že automat A_1 rozpoznává všechna slova, která mají sudý počet nul (pozn. nula je také sudé číslo). Druhý automat pak přijme pouze slova, u nichž je počet jedniček beze zbytku dělitelný třemi. My máme za úkol sestrojit automat, který bude přijímat jeden **nebo** druhý jazyk. Formálně zapsáno:
$L = L_1 \cup L_2$, kde $L_1 = \{w \in \{0, 1\}^* \mid |w|_0$ je dělitelné 2$\}$ a $L_2 = \{w \in \{0, 1\}^* \mid |w|_1$ je dělitelné 3$\}$.

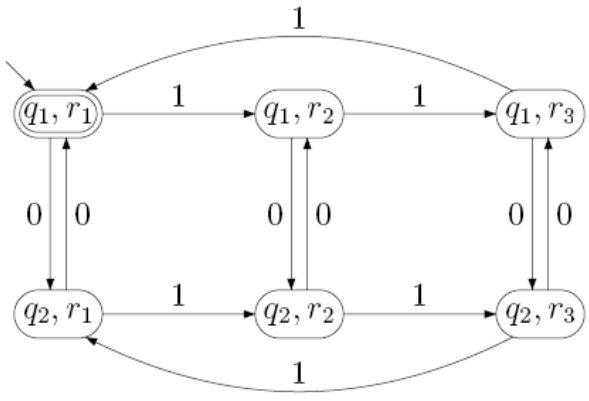
Nyní zbývá určit, jak poznáme, že dané slovo w patří do jazyka $L(A_1) \cup L(A_2)$. Řešení je jednoduché. Slovo w zpracujeme postupně prvním i druhým automatem, a pokud je alespoň jedním z nich přijato, pak je přijato i výsledným sjednocením.

Pokud nyní všechny tyto poznatky aplikujeme na výše uvedený příklad automatů A_1 a A_2, tak dostaneme sjednocení, které přijímá slova se sudým počtem nul nebo s počtem jedniček dělitelným třemi. Výsledný automat je na obrázku 7.

Velice podobná je konstrukce průniku. Všechny body postupu jsou stejné, až do vyhodnocování přijímacích stavů. Tentokrát se nejedná o disjunkci jazyků, nýbrž o jejich konjunkci. Aby byl stav přijímající ve výsledném automatu, musí být přijímající i v **obou automatech zároveň**. Logicky tedy bude mít stejný počet stavů jako u sjednocení, ale stavů koncových bude počet shodný nebo (častěji) nižší. Průnik automatů A_1 a A_2 je znázorněn na obrázku 8 a jak je patrné, přijímající stav už je pouze jeden jediný, v tomto případě společný počáteční.



Obr. 7 – Sjednocení automatů A_1 a A_2



Obr. 8 – Průnik automatů A_1 a A_2

3.6 Nedeterministický konečný automat

Doposud jsme měli v každém stavu automatu vždy jasně definováno, kam se dostaneme daným symbolem. Přechody byly definovány pro všechny stavy všemi symboly. Někdy ale při konstrukci automatu můžeme potřebovat něco jako rozhodování. Jednou bychom chtěli jít znakem do stavu jedna, podruhé ale tím samým znakem do stavu dvě.

Tento **nedeterminismus** nebyl v automatech, jejichž definici jsme si uvedli dříve, vůbec povolen. Jeho zavedením získáme mírnější restrikce při vytváření automatů. Jedním symbolem je možné přejít do více stavů nebo v případě opačném nemusí být přechod zadán vůbec. Toto je první zásadní rozdíl, kterým se liší **zobecněný nedeterministický konečný automat** (ZNKA) od deterministického. Druhou zvláštností je možnost zvolit libovolný počet počátečních stavů.

ZNKA je tedy uspořádaná pětice $A = (Q, \Sigma, \delta, I, F)$**,** kde

- Q je konečná neprázdná množina stavu,
- Σ je konečná neprázdná množina zvaná vstupní abeceda,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ je nedeterministická přechodová funkce,
- $I \in Q$ je neprázdnou množinou počátečních stavů a
- $F \subseteq Q$ je množina přijímajících (koncových) stavů.

Jak ale rozhodneme, zdali bylo zadané slovo nedeterministickým automatem přijato či nikoli? Jednoduše. Existuje-li alespoň jedna cesta orientovaným grafem taková, že průchod končí v koncovém stavu, pak je i celé slovo automatem přijato. Ostatní cesty přitom nemusíme brát v úvahu. Na rozdíl od deterministického automatu může existovat i nekonečně mnoho různých průchodů automatem jedním slovem, což bývá často zrádné a není pak lehké najít tu pravou, která končí ve stavu přijímajícím.

3.7 Převod nedeterministického automatu na deterministický

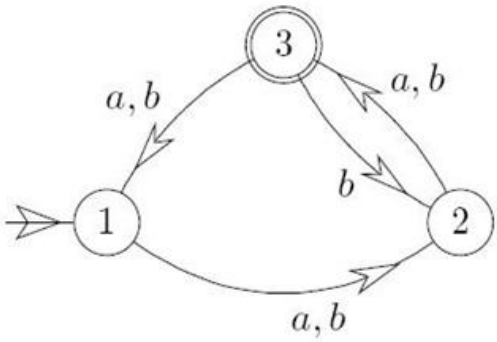
Po předchozím definování zobecněného nedeterministického konečného automatu vyvstává otázka, v čem může být prospěšnější vytvářet automat s nejednoznačnými přechody a více počátečními stavy, když to manipulaci rozhodně nijak nezjednoduší. Důvod je ale zřejmý. Vytvořit ZNKA je mnohdy o poznání lehčí. Nejdůležitější na tom všem je ale fakt, že každičký nedeterministický konečný automat lze algoritmicky převést na ekvivalentní deterministický.

Při konstrukci deterministického automatu je třeba si uvědomit, že výsledný automat může mít až exponenciálně více stavů. To ale naštěstí většinou nemá, protože se při převodu nezabýváme nedosažitelnými stavy.

Samotný postup konstrukce je pak následující:

1. Začneme se stavem reprezentujícím množinu I počátečních stavů nedeterministického automatu A.
2. Dokud máme v sestrojovaném automatu A′ stavy s nadefinovanými přechody, vybereme si jeden takový q a znak x. Pro všechny stavy reprezentované q najdeme všechny možnosti přechodu znakem x v A a shrneme je v nové množině stavů q' (ta již v našem automatu může být sestrojená).
3. Když nový stav reprezentuje množinu, která protíná nebo ze které se v A dá dostat ε-přechody do F, označíme jej jako přijímající.

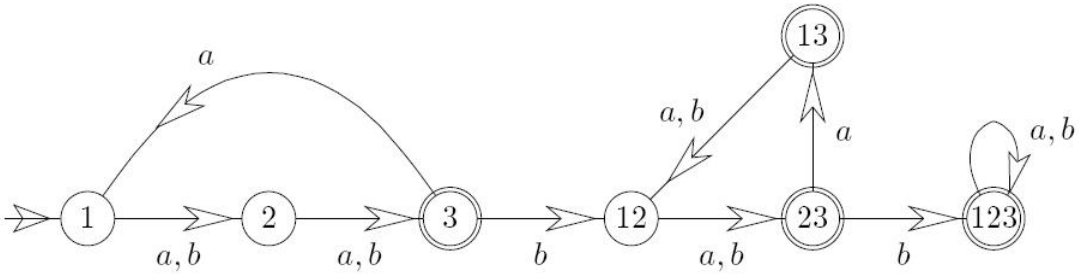
Opět bude nejlepší, když si tento krkolomně popsaný algoritmus ukážeme na konkrétním jednoduchém příkladu. Poslouží nám k tomu automat z obrázku 9. Jeho nedeterminismus spočívá přitom pouze v tom, že se ze stavu tři můžeme znakem b dostat do dvou různých stavů.



	a	b
→ 1	2	2
2	3	3
← 3	1	1, 2

Obr. 9 – Nedeterministický konečný automat s přechodovou tabulkou

Jestliže postupujeme přesně podle výše popsaného postupu, začneme konstrukci v množině počátečních stavů {1}. Z této množiny (pouze stav jedna) doplníme přechodovou tabulku pro oba symboly. Stejný postup i pro druhý stav. Zatím se tabulka od zadání v ničem neodlišuje. Rozdíl přijde až u písmena b ve třetím stavu, jak jsme naznačili dříve. Zde musíme do patřičného políčka vepsat množinu {1, 2}. To znamená, že v dalším kroku zapisujeme přechody jak ze stavu jedna, tak ze stavu dva. Tímto postupem se nám podařilo sestrojit automat na obrázku 10. Pro zjednodušení zápisu množiny stavů {1, 2, 3} vepisujeme do kroužků tyto údaje bez závorek a čárek, tedy pouze 123.



Obr. 10 – Graf výsledného deterministického automatu

Přechodová tabulka výsledného převedeného deterministického konečného automatu tedy vypadá takto. Jak je patrné z obrázku, přibyly dokonce čtyři nové stavy. Všechny stavy, které mají ve své množině přijímající vrchol tři, jsou pak také samy koncové.

	a	b
→ 1	2	2
2	3	3
← 3	1	1, 2
1, 2	2, 3	2, 3
← 2, 3	1, 3	1, 2, 3
← 1, 3	1, 2	1, 2
← 1, 2, 3	1, 2, 3	1, 2, 3

4 Softwarový proces projektu Automat

Diplomová práce by měla ukázat, co všechno se student za léta strávené na univerzitě naučil a jak tyto vědomosti umí reálně aplikovat. Vývoj softwarového díla, na kterém celá závěrečná práce stojí, si o to přímo říká. Nejedná se totiž o program, který student napíše přes víkend. A pokud náhodou ano, pak je buď génius, nebo ta práce podle toho vypadá a nemá ji šanci obhájit.

Katedra informatiky a výpočetní techniky ale během celých pěti let studia nabízí širokou škálu předmětů zabývajících se problematikou návrhu a vytváření softwarových systémů. Toto odvětví se obecně nazývá softwarové inženýrství. Základem všeho je přitom softwarový proces, který zahrnuje postup činností a celou řadu metod nutných k tvorbě konečného projektu. Proces je složen z mnoha části, spojených buď tematicky, nebo logicky.

Obecná definice, naznačená v předchozím odstavci, tedy zní takto:

Softwarový proces je po částech uspořádaná množina kroků směřujících k vytvoření nebo úpravě softwarového díla.

Tento popis si zasluhuje podrobnější vysvětlení. Krokem máme na mysli konkrétní aktivitu, ale i jiný podproces. Oboje přitom může probíhat současně, proto je nutná nějaká správa a koordinace. Asi nejdůležitější vlastností softwarového procesu je jeho univerzálnost. Tím, že jej lze aplikovat na více podobných produktů, zajistíme jeho znovupoužitelnost, což zvyšuje kvalitu i efektivitu.

4.1 Základní typy softwarového procesu

Ačkoli dodnes neexistuje jednoznačný referenční model softwarového procesu (ikdyž jeden z nich, uvedený dále v textu, k tomu má již blízko), můžeme přesto říci, že většina vychází z **modelu vodopádového**. Je to nejintuitivnější metoda vývoje, kterou často i nevědomky používá od začátku každý vývojář a programátor. Nevyžaduje žádné hlubší znalosti problematiky, sled událostí je logický. Jedna fáze plynule navazuje na druhou. Celkem jsou tyto čtyři: analýza požadavků a jejich specifikace, návrh softwarového systému, implementace (kódování) a na konec testování a udržování vytvořeného produktu.

Tento model má ovšem hned několik závažných nedostatků. Žádná fáze (kromě první) nemůže započít dříve, než je předchozí fáze dokončena. Je to z toho důvodu, že výsledky jedné fáze slouží jako vstupy do té následující. Dalším problémem je časová náročnost vývoje projektu. Od chvíle, kdy byly analyzovány požadavky, až po předání produktu probíhá práce vždy jen v jediném podprocesu, nikdy ne souběžně ve dvou, natož ve více z nich. Pak je také potíž v tom, že výsledný produkt je zcela závislý na kvalitě specifikování požadavků, protože dílo v konečné podobě se k zadavateli dostane až na konci procesu jako hotový celek. Případné nedostatky budou tak odhaleny pozdě a jejich oprava bude opět časově velice náročná. V horším případě může mít i katastrofální důsledky, kdy se bude muset přeprogramovat třeba i samotné jádro systému.

U menších projektů bývá však tento model stále využíván pro svou jednoduchost. Jakýkoli složitější paralelismus v rámci aplikace je také vyloučen, pokud na ni pracuje pouze jeden člověk. V takovém případě je lepší aspoň takto řízený proces, než žádný.

Mnohem efektivnější je však využít **iterační způsob vývoje**. Celý proces je rozdělen do několika částí, které mají podobnou strukturu. Jedna iterace spočívá v tom, že se splní jistá množina požadavků, která se naimplementuje a předá k testování. Mezi tím začne vývoj další iterace a tak stále dokola, až jsou splněny všechny požadavky a projekt je hotový. Důležité je, že na konci každé iterace vzniká spustitelný kód. Takto se odhalí případné chyby (ať už z analýzy či kódování) mnohem dříve a náklady na jejich odstranění jsou mnohonásobně menší. Navíc má zadavatel projekt stále pod jakousi kontrolou a může sledovat, jak na něm práce probíhají.

Iterační způsob vývoje je jedním z pilířů procesu RUP. Ten byl také použit pro tuto diplomovou práci a podrobně si jej popíšeme hned v následující kapitole.

4.2 RUP

Proces RUP (Rational Unified Process) je výsledkem úsilí mnoha významných firem, přičemž hlavní iniciativu vyvinula společnost Rational (proto její jméno v názvu). Cílem RUP je vyvinout produkt, který v co největší míře odpovídá požadavkům zákazníka, přičemž je dodán v rozumném čase a za přijatelné prostředky.

Výhod oproti vodopádovému modelu je hned několik. Kromě iteračního způsobu vývoje, kterému jsme se věnovali v předchozí části, je to i neustálá kontrola a správa požadavků. Všechny změny jsou monitorovány a dokumentovány. Aplikace je navíc neustále konzultována se zákazníkem. Ten může průběžně vývoj ovlivňovat a opravovat dle svých představ. Nemůže se tak stát, že po několika měsících (někdy i letech!) vývoje nedojde k předání software, protože zadavateli výsledek nevyhovuje.

Dalším znakem RUP je používání již existujících částí kódu a komponent. To je známkou vyspělosti tvůrců software, protože ti vysledovali prvky znovu použitelnosti nejen v produktech svých, ale i v komponentách volně dostupných.

Nespornou výhodou je také možnost vizualizace fází procesu. Usnadňuje komunikaci mezi různými aktéry. Názorně a detailně, ale přitom neodborně vysvětluje často i graficky procesy, algoritmy, strukturu i chování systému. Standardem pro vizualizaci zmiňovaných celků je firmami uznávaný jazyk UML (Unified Modeling Language). Jak již vyplývá z anglického názvu, primárně slouží k srozumitelnému modelování softwarových systémů. Pro tento účel poskytuje celou řadu diagramů. Některé popisují statickou strukturu, jiné dynamické chování systému. O tom si však konkrétně povíme později.

Vlastní proces RUP je složen z několika toků činností. Jmenovitě to jsou byznys modelování, specifikace požadavků, analýza a návrh, implementace, testování a nasazení. Je možné mezi ně započítat také manažerské podpůrné toky, jako řízení změn a konfigurací, projektové řízení a správu prostředí. Co je však hlavní, všechny toky mohou probíhat současně! Intenzita každého z nich bude ovšem jiná v závislosti na tom, v jaké fázi se projekt nachází.

V následujících kapitolách si podrobně probereme většinu z toků a ukážeme si, jakou úlohu hrály a jak byly aplikovány při tvorbě projektu Automat, součásti diplomové práce.

4.3 Specifikace požadavků

Než se dostaneme k samotné specifikaci požadavků, považuji za vhodné zmínit, proč jsem přeskočil fázi byznys modelování. Tato fáze totiž popisuje strukturu a dynamiku společnosti či organizace. Má za úkol vytvořit a popsat modely podnikových procesů. Vlastními slovy řečeno má srozumitelně pro obě strany popsat problematiku, kterou se podnik (zadavatel) zabývá. V případě diplomové práce však tato fáze ztrácí smysl. Procesy školy nás až tolik nezajímají a na softwarové dílo bude mít vliv pouze látka teoretické informatiky, která byla podrobně popsána ve třetí kapitole textu.

Cílem specifikace požadavků je popsat funkcionalitu softwarového systému. Modely specifikace požadavků slouží k odsouhlasení zadání mezi vývojovým týmem a zadavatelem. Patří mezi ně hlavně dva diagramy UML – diagram případů užití a sekvenční diagram. Součástí fáze je rovněž několik artefaktů, mezi ty hlavní patří dokument s přesným zadáním (Stakeholder Requests), dokument vizí (Visions) a slovník pojmů (Glossary).

Z těchto dokumentů se hned v další kapitole budu věnovat hlavně přesnému zadání. Vize aplikace jsou totiž jasné a vystihuje je především zadání DP. To se samozřejmě musí splnit co nejvýstižněji, přičemž rozšířená nadstavbová funkcionalita bude jen k dobru. Z tohoto hlediska je mnohem důležitější, aby si zadavatel a vývojář navzájem dobře rozuměli. Proto je potřeba zavést slovník pojmů, kde jsou zákazníkovi popsány věci technické a naopak zhotoviteli je vysvětlena terminologie oblasti zaměření projektu. Slovník pojmů připomíná tak trochu seznam zkratek ze začátku textu, jen je použito hlubšího a podrobnějšího popisu. Dokument je přiložen v elektronické podobě na nosiči CD v adresáři Dokumentace spolu s ostatními artefakty.

4.3.1 Dokument Stakeholder Requests

Jak již bylo zmíněno, tento dokument má za úkol co nejlépe specifikovat zadání projektu. Přesněji řečeno, dokument popisuje všechny možné typy požadavku zadavatele. Tím může být zákazník, koncový uživatel nebo univerzita zastoupena vedoucím DP. V případě této diplomové práce bylo na začátku pevně dáno pouze téma, které ale už samo o sobě nebylo jen povrchní. Nakonec ale byla s vedoucím DP dohodnuta spousta konkrétních detailů a konečnou podobu oficiálního zadání si můžete přečíst na straně dvě. Skládá se celkem z devíti hlavních požadavků, které v následujících odstavcích podrobněji specifikujeme tak, jak bylo dohodnuto s vedoucím.

Prvním bodem byla definice konečného automatu. Je tím myšleno, že si bude uživatel moci zadat a definovat automat libovolných vlastních rozměrů nad abecedou dvou nebo i tří písmen. Samozřejmě musí mít automat rozumný počet stavů, který bude po dohodě omezen na maximálních osm či deset (nakonec byl zvolen počet dvanácti stavů). Samozřejmostí pak je

možnost zvolit mezi automatem deterministickým a nedeterministickým. Pod definici automatu spadá také načtení již definovaného automatu ze souboru.

Pro vytvoření konkrétního automatu potřebujeme znát počet stavů a písmen abecedy. Na základě těchto údajů bude vygenerována přechodová tabulka, stejně jako graf automatu. Na uživateli nyní záleží, zdali přechody samotné zadá pomocí klávesnice do tabulky či nakliká myší do grafu. Oba postupy by mělo být možno vzájemně kombinovat, změny v tabulce se projeví v grafu a naopak, to buď po stisku příslušného tlačítka, nebo lépe automaticky při každé provedené změně. Pokud byl automat načten ze souboru, mělo by být možno jeho přechody editovat. Typ automatu, počet stavů a písmen abecedy přitom zůstává.

Dále převod definovaného automatu do normovaného tvaru musí probíhat korektně, tento algoritmus by měl být viditelný (znázorněn přechodovou tabulkou i graficky) a krokovatelný vpřed i vzad.

Každému zadanému automatu bude také možno předhodit slovo nad danou abecedou a graficky bude realizován průchod automatem po jednotlivém čteném znaku, samozřejmě opět s popisem a s konečným vyhodnocením, zdali bylo slovo automatem přijato či nikoli. Zde je vhodné, aby bylo možno procházet zároveň jak zadaným (jedním i dvěma!), tak výsledným automatem.

Program bude umožňovat zadat také druhý automat, který posléze poslouží k operacím sjednocení a průniku dvou deterministických automatů. Tyto operace budou znázorněny jak grafem, tak tabulkami.

Názorně, korektně a detailně bude možné převést nedeterministický konečný automat na deterministický. Opět by mělo být možno postup krokovat vpřed i vzad.

Taktéž minimalizace konečného automatu by měla být podrobně popsaná a postup algoritmu v tabulce krokovatelný. Zde bude názornost složitější, protože se bude vytvářet několik tabulek. Mělo by tak být umožněno procházení alespoň těch. V lepším případě pak zajistit vysvětlení tvorby každé této tabulky zvlášť.

Poslední požadavek se týká persistence zadaných dat. Definovaný automat by mělo být možno uložit ve tvaru ještě před zpracováním a poté znovu před zadáním automatu načíst. Tím by se zjednodušila opětovná manipulace s ním. Zároveň bude možno rychle porovnávat výsledky různých výpočtů. Vyskytne se tak možnost připravit si spoustu ukázkových příkladů pro demonstrativní účely. Stejně užitečná by byla funkce pro uložení automatů, které vznikly všemi výše zmiňovanými algoritmy. Tím bude umožněno pracovat s výslednými automaty dále. Například převedený nedeterministický automat můžeme dále zkoumat, převést do normovaného tvaru (ve kterém se ale už bude nacházet ihned po převodu, což si ale můžeme alespoň ověřit), minimalizovat, ale hlavně jej použít pro operace sjednocení a průniku.

Po prvním přezkoumání primárních požadavků je seznam průběžně doplňován volitelnými a upřesňujícími návrhy, které by ve výsledku měly (nebo mohly) být přínosem. Tyto jsou většinou nepovinné a pouze doporučené vedoucím diplomové práce.

Velmi zajímavá by byla možnost výsledky operací a algoritmů ukládat na disk, tedy exportovat do souborů. To bude vhodné především v případě manipulace s velkými automaty, které nepůjde zobrazit graficky. Soubor by také byl výchozím vstupem pro tisk, jelikož výsledky a postupy bude uchovávat ve srozumitelném tvaru.

Přínosem by také nepochybně byla lokalizace do anglického jazyka, případně návrh aplikace tak, aby bylo jednoduše možno provést lokalizaci do libovolného jazyka.

Aby měla aplikace svůj smysl, bylo vy vhodné zobrazovat co nejvíce komentářů a nápovědných textů přímo při běžné práci, například vyskakující okénka, informativní hlášky a vestavěná nápověda, třeba po najetí kurzorem nad patřičný objekt.

Poslední požadavek by kosmeticky upravoval vzhled aplikace. Mohlo by jít přejmenovat stavy i písmena abecedy. Práce by pak byla uživatelsky přívětivější a přizpůsobitelnější pro každého, kdo by program spustil. Je totiž individuálním zvykem každého z nás si písmena abecedy pojmenovat po svém, stejně jako stavy. Někdo preferuje latinskou abecedu, jiný dá přednost číslům. Ani tento požadavek ovšem nemá vliv na funkčnost.

Toto byl výčet všech oficiálních požadavků zadavatele před zahájením návrhu a kódování aplikace. Abych to shrnul – všem hlavním požadavkům na funkčnost bylo do posledního detailu vyhověno. U některých došlo dokonce i k mírnému nadstandardnímu splnění. Stavů tak lze například nadefinovat až dvanáct, při vytváření automatů lze přidat další stav, nebo přidat i ubrat třetí písmeno. Doplňující zadání bylo zakomponováno také, ale některé požadavky byly po dohodě s vedoucím z implementace vyloučeny. Uživatel tak má možnost přepnout celé prostředí a aplikaci do anglického jazyka, výsledky algoritmů lze uložit do souboru a nápovědných i popisných textů je také dost. Často na vás vyskočí také nějaké to informační okénko. Jelikož ale byla především lokalizace do angličtiny dosti rozsáhlá a časově náročná, upustilo se od požadavku přejmenování stavů a písmen abecedy, které by vývoj značně zpomalily. Také některé interaktivní nápovědy při pohybu kurzoru myši nakonec nebyly do programu zahrnuty, protože testy ukázaly, že snímání pozice kurzoru se zaregistrovanou akcí na jednotlivých komponentách značně vytěžují procesor a celkově zpomalují chod aplikace. To lze ale považovat pouze za malé nedostatky, které výslednou kvalitu produktu nijak výrazně nesnižují.

4.3.2 Diagramy případů užití

Tato kapitola poskytne jeden z nejlepších a nejpřehlednějších popisů aplikace z pohledu uživatele či čtenáře tohoto textu, aniž by musel spouštět samotný program.

Definice případů užití zní následovně:

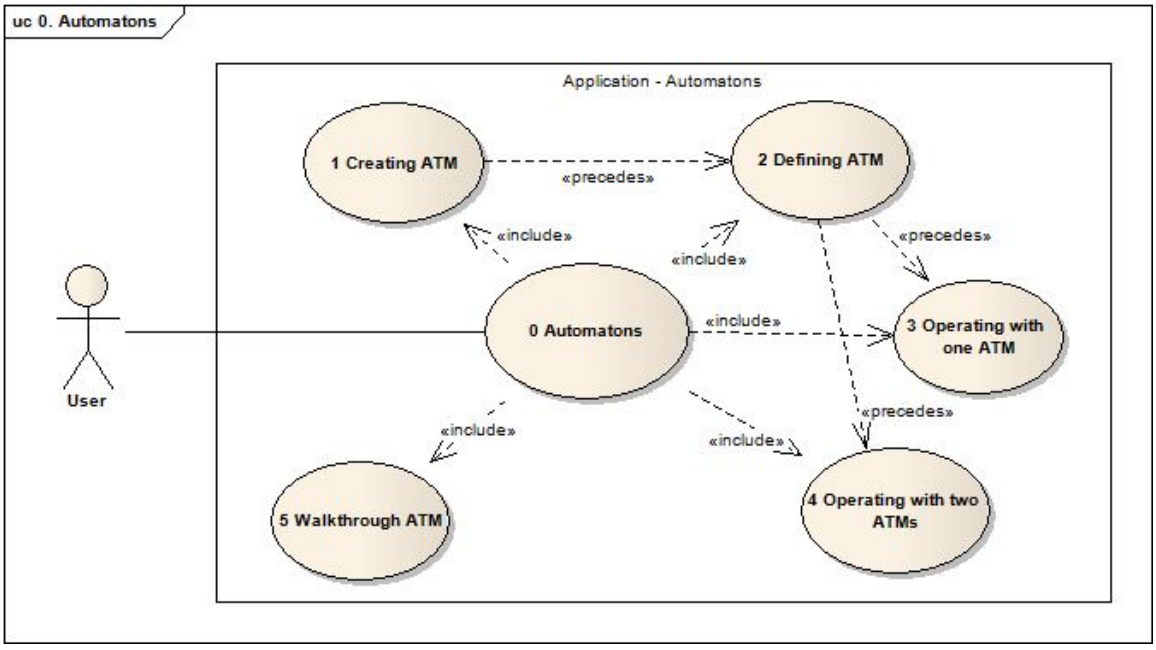
Případy užití specifikující vzory chování realizovaných softwarovým systémem. Každý případ užití lze chápat jako posloupnost vzájemně navazujících transakcí vykonaných v dialogu mezi aktérem a vlastním softwarovým systémem.

Diagram případu užití pak popisuje vztahy mezi aktéry a jednotlivými případy použití. Obecně lze tedy říci, že účelem je definovat, co existuje vně vyvíjeného systému a co má být systémem prováděno. Diagram je vlastně takový náhled na celý projekt (i jeho jednotlivé části) jako na černou skříňku, kdy zákazníka zajímá, **co** se má udělat, ale už mu je úplně jedno, **jak** to bude zajištěno vnitřně, aplikačně. Diagram se tedy skládá z aktérů (v našem případě jeden obecný uživatel), případů užití (jednotlivé akce a operace) a vztahy mezi nimi.

Tyto vztahy, též relace, se dále dělí na použití (uses), rozšíření (extends) a generalizaci (vyznačeno orientovanou šipkou s prázdným trojúhelníkem na konci). Relace použití vyjadřuje situaci, kde určitý scénář popsaný jedním případem užití je využíván i jinými případy užití. Generalizace (zobecnění či specializace) vyjadřuje vztah mezi obecnějším případem užití a jeho speciálním případem. A relace rozšíření logicky jen rozšiřuje jiný případ užití nebo k němu nabízí jiné varianty provedení.

Pro potřeby této DP chápeme diagram případů užití tak, že zachycuje systém z pohledu uživatele. Jedná se o popis chování produktu tak, jak reaguje na požadavky přicházející zvnějšku aplikace. Zobrazuje interakci mezi primární rolí a systémem samotným, který je reprezentován sekvencí jednoduchých kroků. Rolí může být uživatel nebo někdo (i něco) jiného, jako administrátor, jiný systém či nějaký hardware. Každý diagram použití se skládá z kompletní skupiny událostí, jak je vidí uživatel nebo aktér.

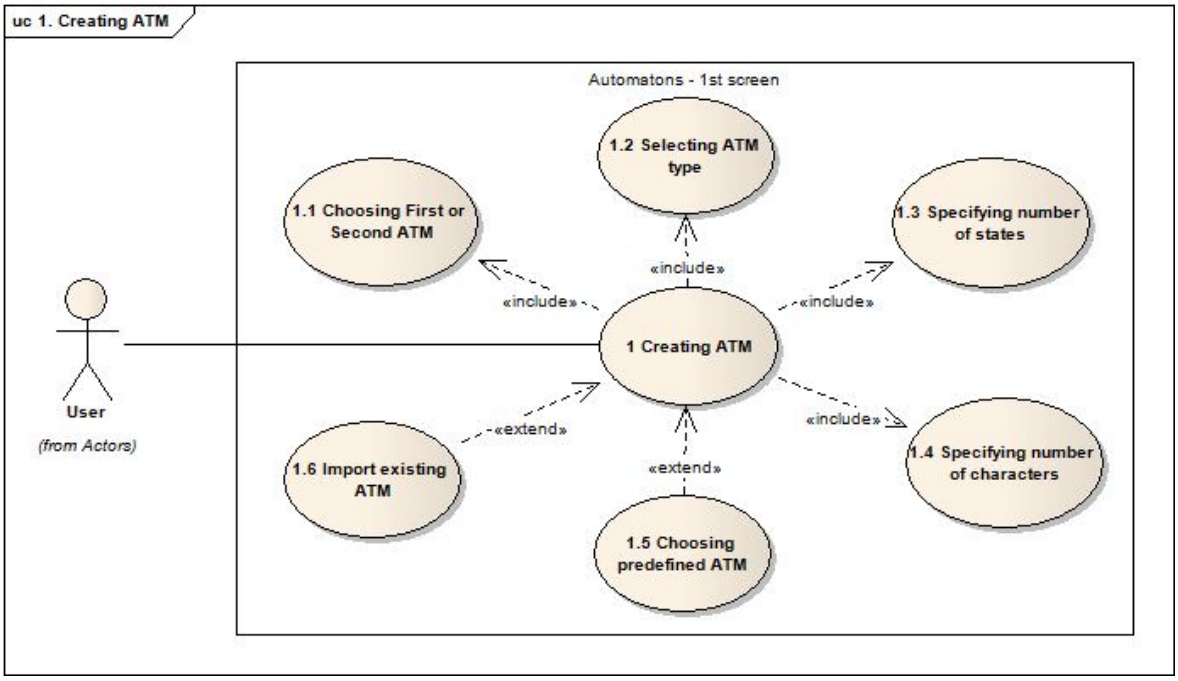
Základní případ užití (obrázek 11) pro projekt Automat definuje hlavní činnosti a možnosti aplikace. Uživatel na program nahlíží jako na celek (*0. Automatons*) – funkčnost je před ním odstíněna. Nemá tak ponětí, jak se konkrétní úkony provádějí na úrovni logiky a kódu. Vytvoření automatu (*1. Creating ATM*) je možné z hlavní nabídky a obsahuje zvolení typu automatu a zadání počet stavů a písmen abecedy. Bez něj není možno dále v používání aplikace pokračovat. Bezprostředně po vytvoření automatu následuje jeho definice (*2. Defining ATM*), kde se přesně vyspecifikují přechody. Až nyní je možno s automaty dále pracovat. Samotné operace se odlišují podle toho, jestli je zadán jeden (*3. Operating with one ATM*) nebo dva automaty (*4. Operating with two ATMs*). Na konci je také umožněn průchod každým automatem na základě zadaného slova (*5. Walkthrough ATM*).



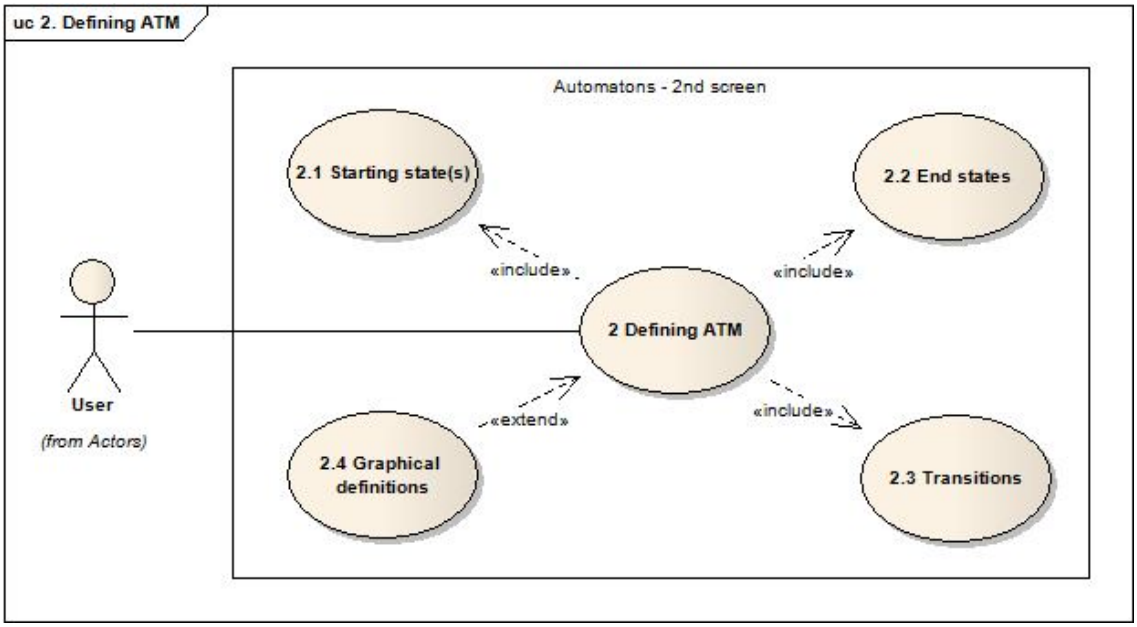
Obr. 11 – Základní případ užití: Automat

K vytvoření automatu je potřeba specifikovat některé základní charakteristiky a právě tyto jsou zachyceny diagramem na obrázku 12. Kromě jiného zahrnuje možnost načtení dříve

uloženého automatu ze souboru. Uživatel si nejprve vybere, který automat zadává, jestli deterministický nebo nedeterministický (*1.2 Selecting ATM type*). Současně, pokud zvolil automat deterministický, volí mezi prvním a druhým (*1.1 Choosing First or Second ATM*). Poté zadá počet stavů (*1.3 Specifying number of states*) a počet písmen abecedy (*1.4 Specifying number of characters*). Pod diagramy *1.5* a *1.6* si lze jednoduše představit načítání automatu ze souboru na pevném disku.



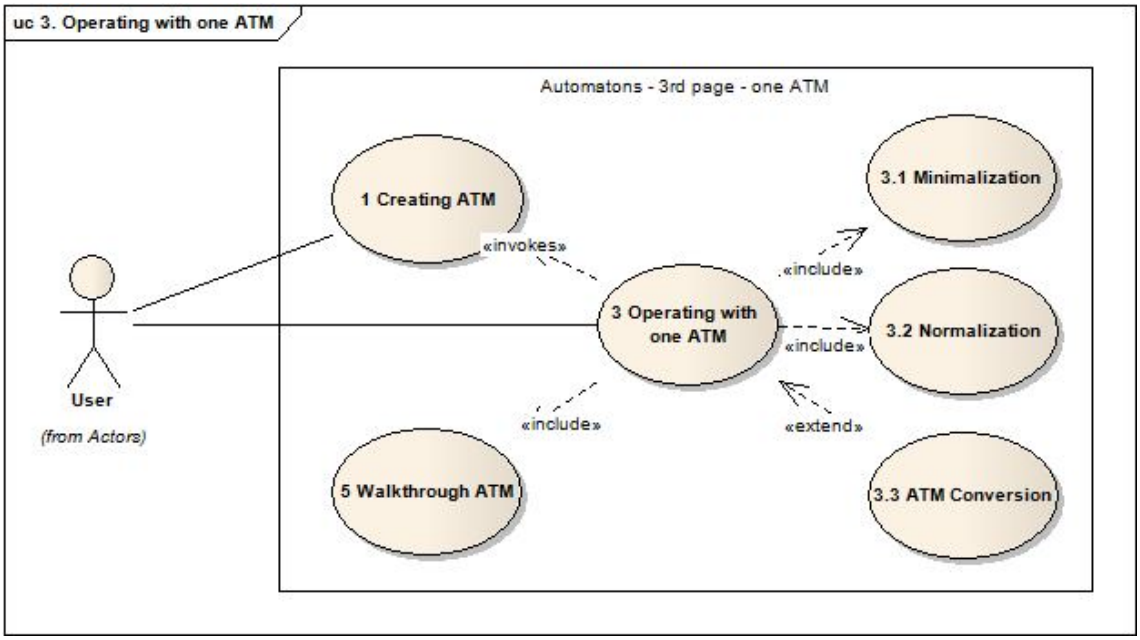
Obr. 12 – Případ užití: Vytvoření automatu



Obr. 13 – Případ užití: Definování automatu

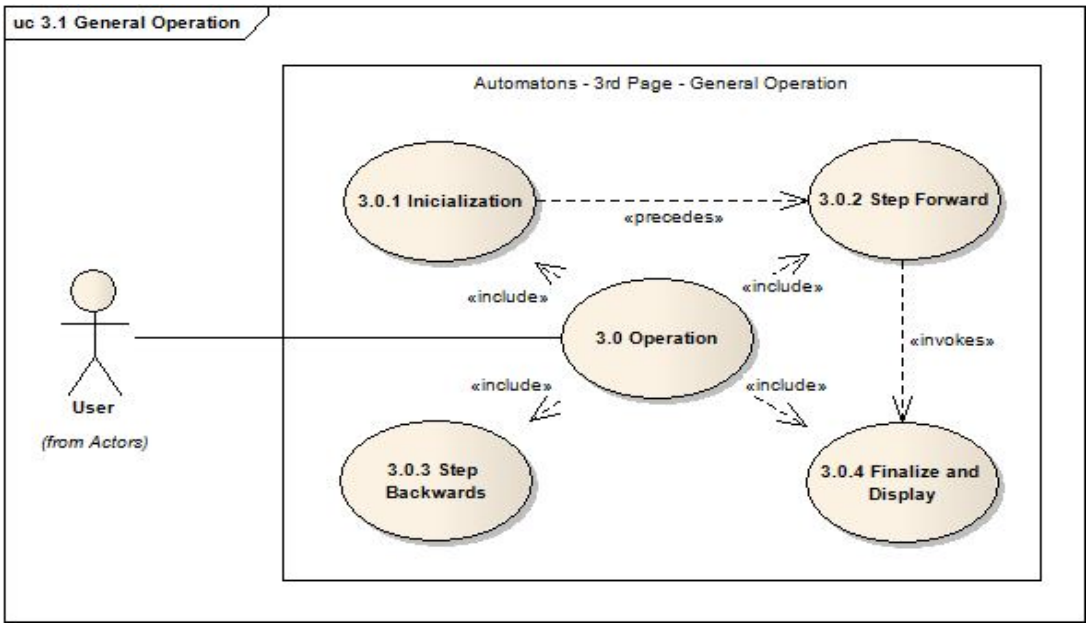
Poté, co jsou zadány parametry automatu, je vytvořena jeho holá kostra. Proto je potřeba specifikovat přechody mezi stavy, jak znázorňuje diagram případu užití obrázku 13. Nejdříve označíme počáteční stav, v případě nedeterministického automatu i více stavů (*2.1 Starting States*). Stejně tak určíme, které stavy budou přijímací (*2.2 End States*). Samotné přechody mezi stavy můžeme definovat dvěma způsoby a to vyplněním přechodové tabulky (*2.3 Transitions*) nebo naklikáním myší do grafu (*2.4 Graphical definition*).

Operace s jedním automatem jsou téměř ekvivalentní pro deterministický i nedeterministický automat a proto jsou zahrnuty pouze v jednom diagramu (obrázek 14). Jakékoli operaci bude rozhodně předcházet vytvoření automatu (*1. Creating ATM*), které je popsáno v jednom z předchozích diagramů. Nejdůležitější pro deterministický automat budou patrně jeho minimalizace (*3.1 Minimalization*) a převod na normovaný stav (*3.2 Normalization*). Naopak pro nedeterministický je hlavní převod na deterministický (*3.3 ATM Conversion*). Oba automaty je ale možno volně procházet na základě zadaného slova (*5. Walkthrough ATM*), diagram bude popsán později.



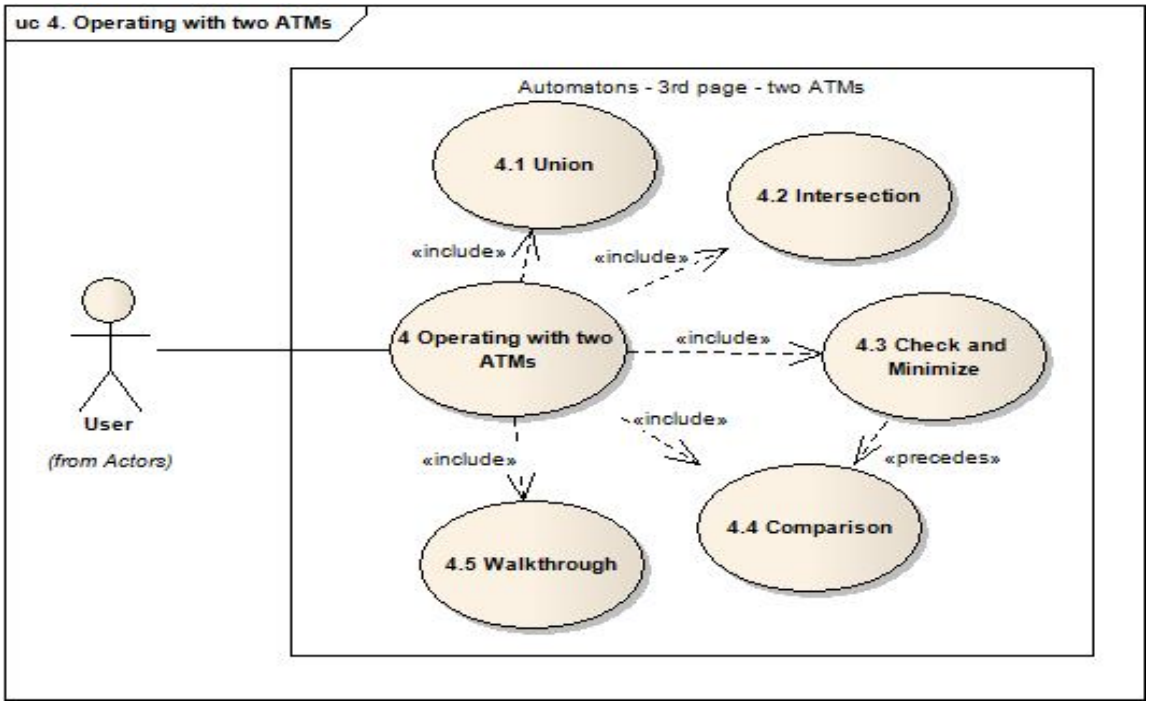
Obr. 14 – Případ užití: Operace nad jedním automatem

Pod obecnou operací, zachycenou obrázkem 15, si lze představit libovolný krokovatelný algoritmus. Mezi ty hlavní patří minimalizace, převod do normovaného tvaru a převod nedeterministického konečného automatu na nedeterministický. Každé operaci musí předcházet jakási inicializace (*3.0.1 Inicialization*). Tou je myšleno nastavení automatu a připravení prostředí pro danou činnost. Po inicializace můžeme provádět kroky směrem vpřed (*3.0.2 Step Forward*) i vzad (*3.0.3 Step Backwards*). Poslední krok vpřed povede k dokončení operace a vyhodnocení (*3.0.4 Finalize and Display*). Jediný rozdíl bude u algoritmu minimalizace, kde je umožněno krokování jak mezi jednotlivými tabulkami, tak v každé z nich samotné (při popisu vytváření).



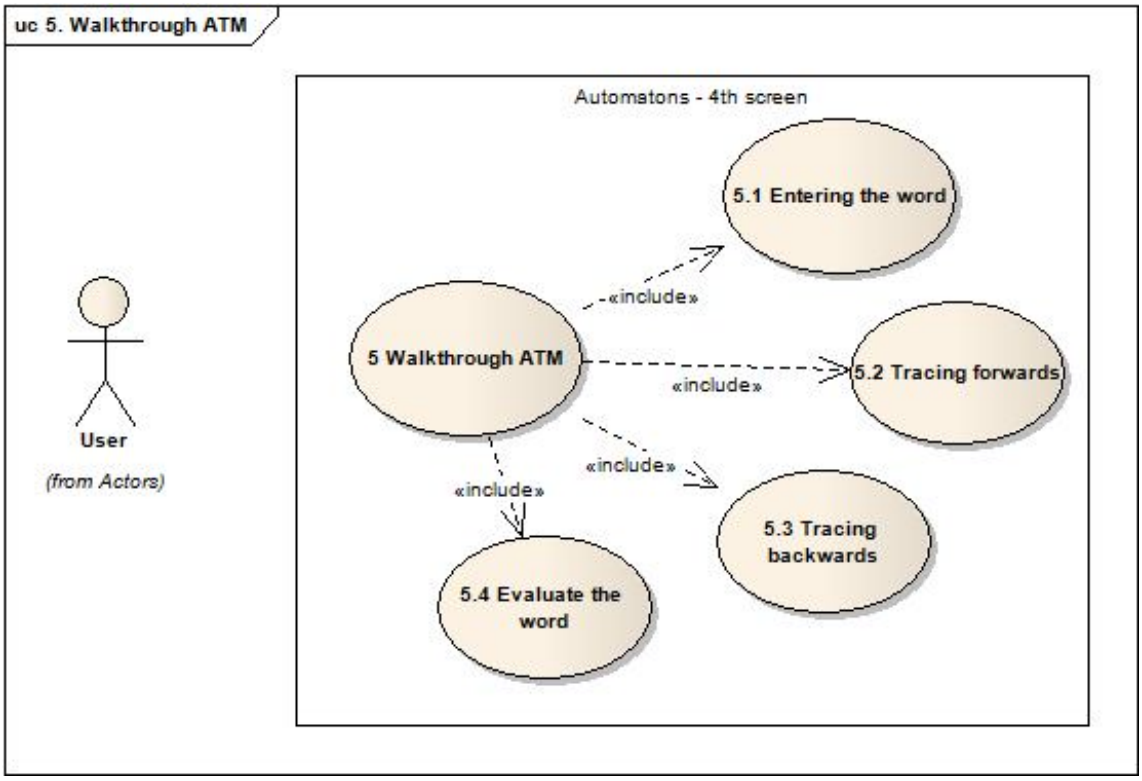
Obr. 15 – Případ užití: Obecná operace s jedním automatem

Operace nad dvěma automaty mohou probíhat pouze tehdy, jsou-li oba deterministické. Všechny tyto operace jsou na obrázku 16. Základními operacemi, které lze provádět se dvěma deterministickými automaty, jsou sjednocení (*4.1 Union*) a průnik (*4.2 Intersection*). Kromě těchto program uživateli nabízí zadané automaty ověřit a minimalizovat (*4.3 Check and Minimize*), aby bylo možné je následně porovnat (*4.4 Compare*). Zajímavá možnost bude srovnat (ne)přijímání zadaného slova všemi třemi automaty (*4.5 Walkthrough*) – dvěma vstupními a jedním nově vzniklým.



Obr. 16 – Případ užití: Operace nad dvěma automaty

Posledním prezentovaným diagramem (obrázek 17) je grafický průchod automatem na základě zadaného libovolného slova. Obzvláště pak porovnávání průchodů automatů před a po operacích bude pro uživatele určitě přínosem. Každý průchod bude začínat zadáním libovolně dlouhého slova (*5.1 Entering the Word*). To se musí skládat pouze z písmen abecedy. Poté může započít samotný průchod vpřed (*5.2 Tracing Forwards*) a vzad (*5.3 Tracing Backwards*), kdy každý přechod bude znázorněn graficky a odlišen barevně. Samozřejmě také bude možno okamžitě (bez průchodu) vyhodnotit (*5.4 Evaluate the Word*), zdali je slovo automatem přijato či nikoli.



Obr. 17 – Případ užití: Průchod automatem

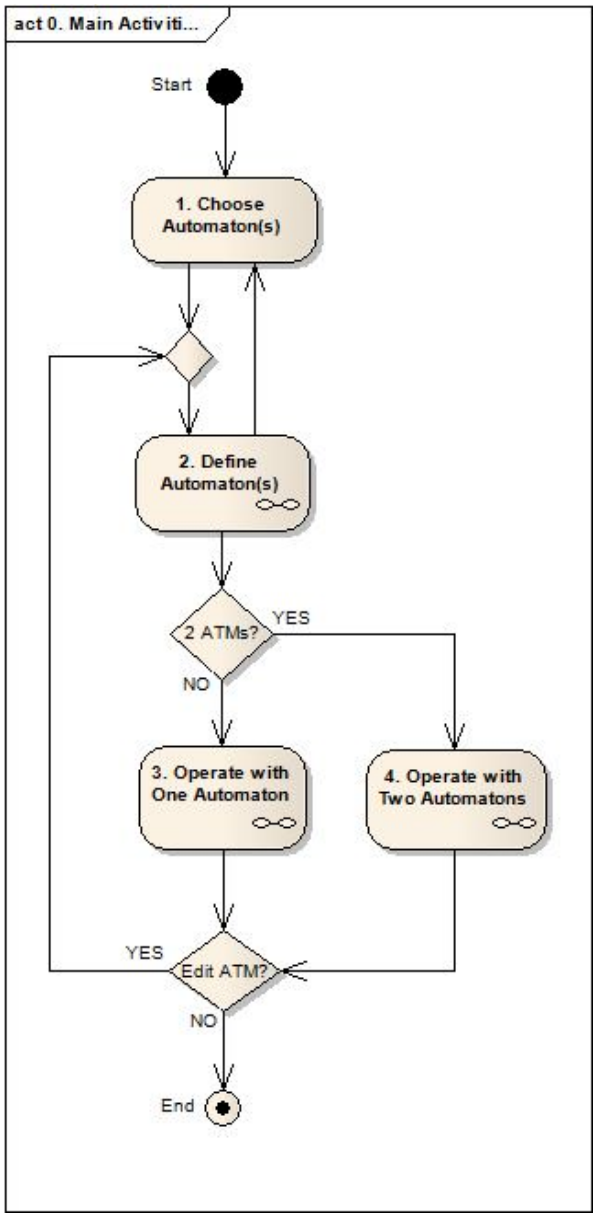
4.4 Analýza a návrh

Úkolem analýzy a návrhu je ukázat, jakým způsobem bude produkt realizován ve fázi implementační. Opět tato fáze využívá několik modelů UML. Vstupními diagramy, které jsou dále zpracovávány, je především dokument požadavků zadavatele a všechny diagramy užití. Neméně důležitý je slovník pojmů, který je součástí všech fází vývoje. Za hlavní výstupní artefakty považujeme stavové a sekvenční diagramy, dále pak diagramy třídní a kolaborační (spolupráce). Sekvenční popisující interakce mezi objekty z hlediska jejich časového uspořádaní, kolaborační z pohledu strukturální organizace objektů. Třídními diagramy se budeme podrobně zabývat až v implementační fázi. Speciální podmnožinou stavových jsou diagramy aktivitní, které si nyní přiblížíme aplikované na projekt Automat.

4.4.1 Aktivitní diagramy

Aktivitní diagram patří k jednomu ze základních nástrojů UML, který definuje sled aktivit při používání aplikace a celkově popisuje chování programu. Proces v diagramu aktivit je reprezentován sekvencí jednotlivých kroků, které jsou zakresleny jako akce (atomické dále nedělitelné kroky) a vnořené aktivity (volání jiných procesů, které mohou být reprezentovány dalším diagramem aktivit). Sekvenci jednotlivých kroků v diagramu aktivit určuje řídicí tok. Rovněž tento model považuji při vysvětlování funkčnosti aplikace za klíčový.

Náš první diagram (obrázek 18) ukazuje všechny hlavní aktivity v logickém sledu, jak jdou za sebou, včetně rozhodování a větvení. Samozřejmě zde nejsou minoritní aktivity jako různé změny nastavení či jazyku, informační okna, nápovědy apod.



Obr. 18 – Hlavní diagram aktivit

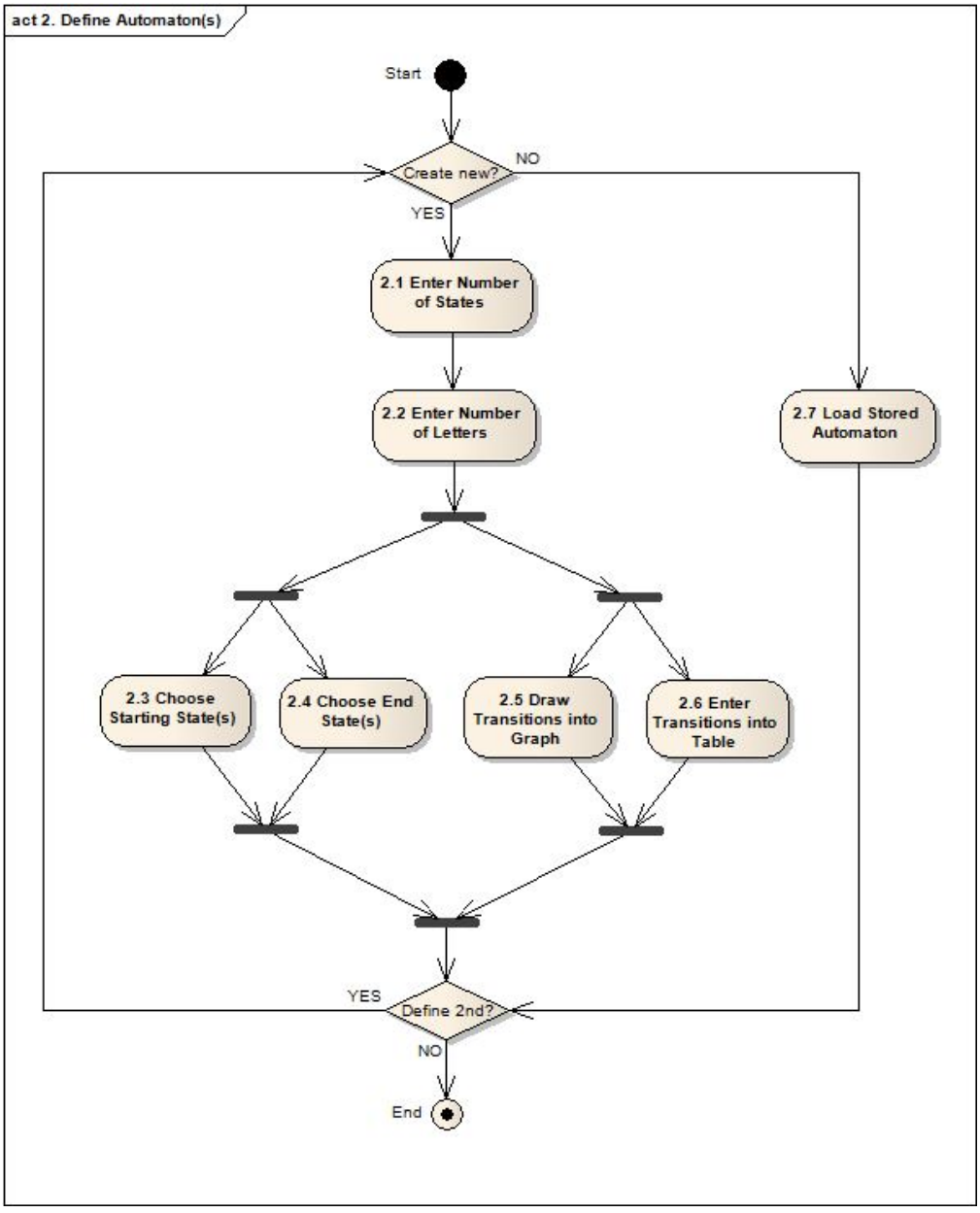
Výběr automatu (1) – tato aktivita pod sebou zahrnuje volbu typu a počtu zadávaných automatů, přičemž si své rozhodnutí můžeme kdykoli rozmyslet a zadat či načíst automat jiný, klidně odlišného typu. K tomu slouží jak menu, tak navigační tlačítka.

Definice automatu (2) – během definice i po ní se lze opět vrátit k zadávání typu automatu, nebo pokračovat k operacím. Samotné definování je popsáno ve poddiagramu, jak naznačuje schéma.

Operace nad jedním automatem (3) – mezi definicí automatu a samotnými operacemi je jednoduchá kontrola podmínky, jak budeme dále pokračovat. Pokud byl při zadání vybrán pouze jeden automat, pak se dostanete do tohoto bodu a veškeré podrobnosti jsou popsány v detailním podschématu.

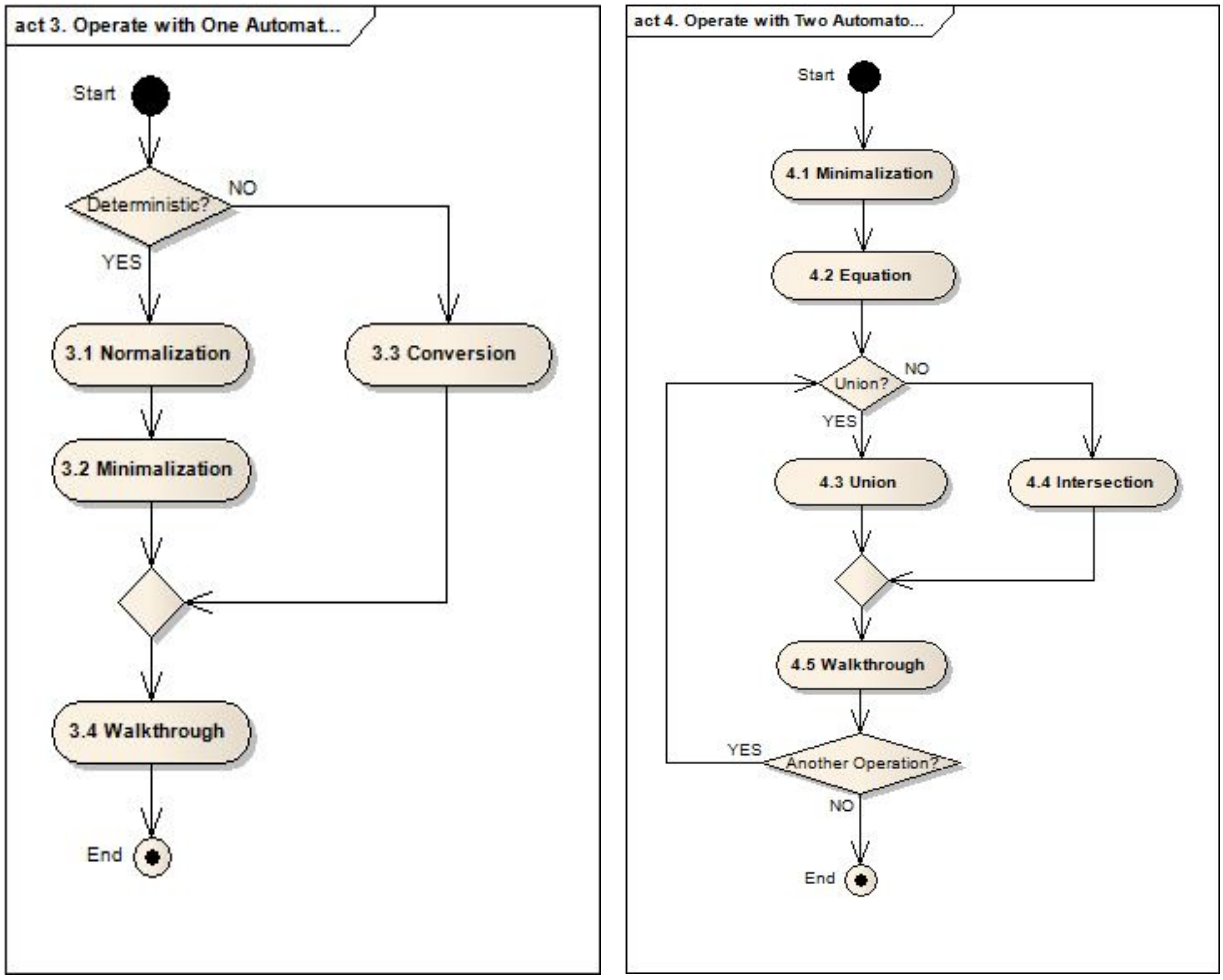
Operace se dvěma automaty (4) – zde se dostaneme druhou větví a opět je detailní popis aktivit této operace popsán podrobnějším schématem.
Než tok aktivit skončí, může se uživatel vrátit na samotný začátek a tedy do kterékoli fáze programu.

V následujícím schématu na obrázku 19 je popsán sled aktivit vedoucí ke kompletnímu definování a zadání libovolného typu automatu. Nejprve je nutné si rozmyslet, jestli chceme automat celý zadat sami, nebo jej jednoduše načteme ze souboru. Pokud se rozhodneme automat načíst, provede se operace nahrání (2.7) a přeskočí se všechny kroky definice. Pokud automat vytváříme sami, pak musíme zadat počet stavů (2.1) a následně počet písmen abecedy (2.2). Pak se cesty dělí na celkově čtyři paralelní větve, které je možno provádět v libovolném pořadí. Jsou to tyto aktivity: vybrání jednoho (i více u nedeterministického automatu) počátečního stavu (2.3), vybrání přijímajících stavů (2.4), zakreslení přechodů do grafu (2.5) nebo zapsání do tabulky (2.6). Definování druhého automatu je ponecháno na rozhodnutí uživatele, stejně jako resetování všech parametrů automatu, překreslení přechodové tabulky nebo jen uložení rozkresleného zadávaného automatu do souboru.



Obr. 19 – Aktivitní diagram: Definování automatu

Operace nad jedním automatem (obrázek 20 vlevo) je složena z těchto aktivit: rozhodování, normovaný tvar (3.1), minimalizace (3.2), konverze (3.3) a průchod (3.4). Rozhodování rozděluje cesty podle toho, je-li automat deterministický či nikoli. Pokud ani, pak je možné provést převod na normovaný tvar a také spustit a procházet algoritmus minimalizace. Nedeterministický automat lze proti tomu jen převést. V obou případech je ale možné procházet zadaným i výsledným automatem na základě vloženého slova.



Obr. 20 – Aktivitní diagramy: Operace s jedním (vlevo) nebo dvěma (vpravo) automaty

Aktivit při manipulaci se dvěma automaty (obrázek 20 vpravo) je malinko více než v předchozím případě. Také je zde minimalizace (4.1), jen v tomto případě souběžně pro oba zadané automaty. Poté proběhne interně jejich porovnání (4.2). Následuje jednoduché rozhodování, jestli přijde na řadu operace sjednocení (4.3) nebo průniku (4.4). Samozřejmě i teď máme možnost procházet automaty po vložení slova (4.5), tentokrát však najednou všemi třemi automaty. Ke kterékoli ze tří posledních operací se lze vrátit a libovolně ji opakovat znovu a znovu.

4.5 Implementace

Mnoho programátorů si zde pomyslí, že se konečně dostáváme k věci a že zde začíná nejhlavnější část práce. Pravdu by měl však jen částečně. V této fázi sice vzniká zdrojový kód, ale v optimálním případě je už kostra vygenerována z předchozích diagramů, protože jazyk UML (hlavně ve verzi 2 a výše) to umožňuje. Zde tedy naplno zhodnotíme aktivitní diagramy, které jsme pracně navrhnuli v předchozí fázi. Téměř polovina úsilí je tedy už dávno za námi. Samozřejmě za nás ale nikdo nevymyslí algoritmy a sofistikovanou logiku. Také statická struktura ještě zatím nebyla vymyšlena, takže pouštět se už do samotného psaní kódu by bylo neuvážené. Naštěstí máme k dispozici ještě dva typy diagramů, třídní diagram a doménový model zobrazující třídy a balíčky jako logické celky.

Úkolem implementační fáze je doplnit navrženou architekturu aplikace o programový kód a vytvořit tak kompletní systém. Toho docílíme jednak psaním holého zdrojového textu, nebo využitím softwarových komponent. Softwarová komponenta je definována jako fyzicky existující a zaměnitelná část systému vyhovující daným rozhraním a poskytující jejich realizaci.

Třídní diagram i doménový model patří do fáze předchozí (konkrétně návrh systému), ale já jsem se rozhodl je použít až v této části, abych mohl popsat strukturu objektů, tříd, komponent a dalších zdrojových souborů.

4.5.1 Třídní diagram

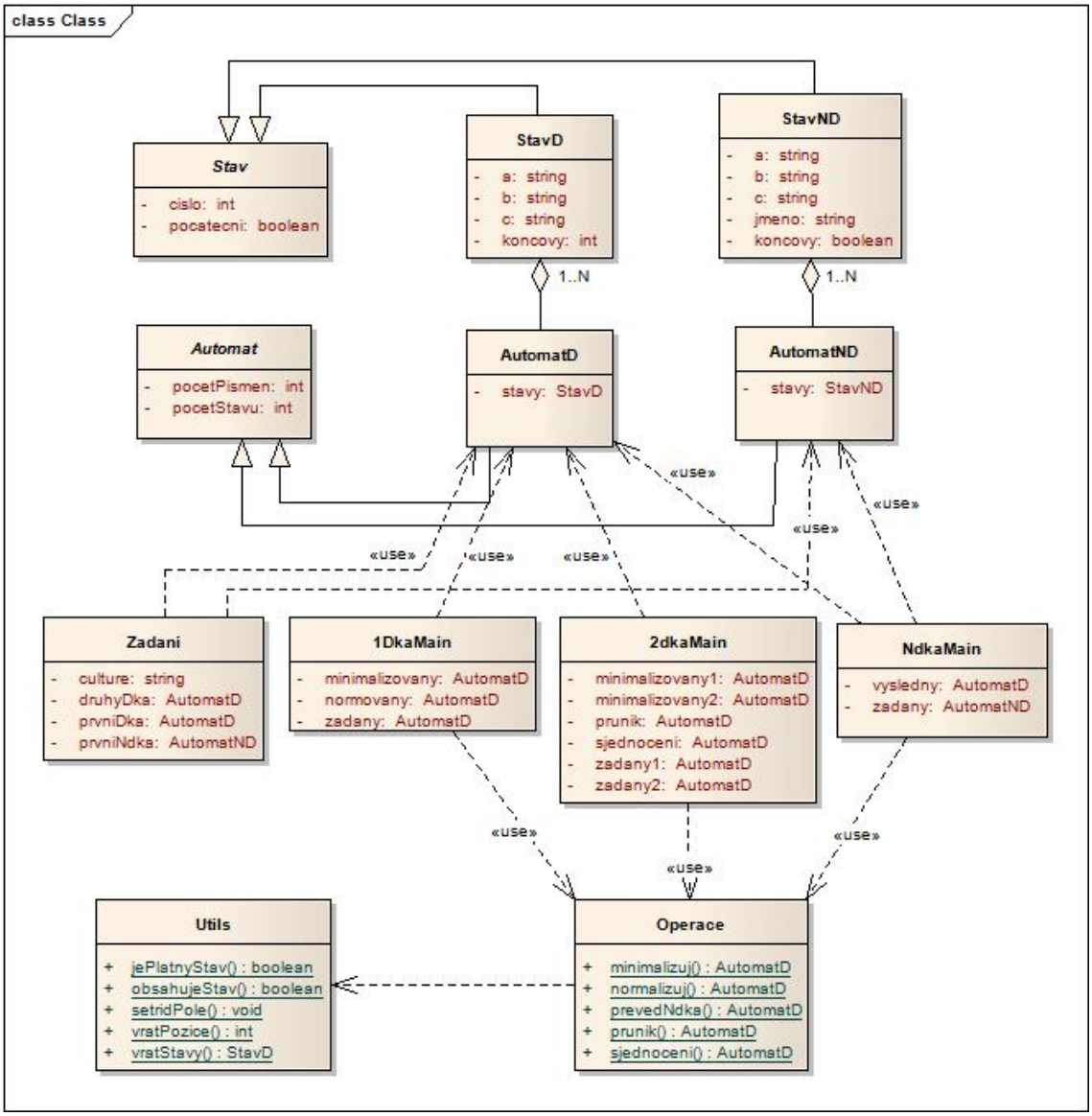
Diagram tříd má za úkol jediné. Co nejpřesněji zachytit statickou strukturu systému. Vzniká už ve fázi analýzy (konceptuální model), upravuje se při návrhu (atributů a operací) a plně jej využíváme právě až k implementaci, kdy tvoříme programový kód.

V diagramu jsou použity dva základní prvky – třídy a vazby mezi nimi. Třída je popis množiny objektů, které mají společnou strukturu, chování a vztahy. Objekt je potom instancí právě jedné třídy. Vlastnosti objektu jsou uloženy v atributech, jeho chování definují metody.

Atribut vypovídá o stavu objektu. Každý objekt má své vlastní, které naplňuje. Výjimku tvoří pouze atributy třídy, jejichž hodnota je sdílena všemi instancemi. Metody určují chování objektu, mohou být reprezentovány funkcí nebo procedurou. Základními metodami jsou konstruktor (vytvoření), gety (čtení), sety (zápis) a destruktor (zrušení). Viditelnost parametrů a operací udává, jak mohou ostatní moduly systému tyto konstrukce využívat. Typy jsou čtyři. Veřejný (public) zajišťuje přístup všem částem systému, dokonce i vnějším. Trošku striktnější je potom viditelnost v rámci balíčku (package, default). Následuje chráněný mód (protected), který se používá při dědičnosti a nejpřísnější je modifikátor soukromý (private). Ten povoluje přístup pouze v rámci třídy.

O struktuře projektu hodně napoví, jaké vztahy mezi sebou třídy mají. Je jich celá řada. Obecnou vazbu mezi objekty znázorňuje asociace (úsečka mezi třídami). Dále jsou to agregace (a silnější kompozice), která vyjadřuje vztah části k celku (prázdný, resp. plný kosočtverec na konci šipky). Dědičnost je znázorněna generalizací (prázdný trojúhelník), závislost přerušovanou orientovanou hranou a rozhraní se implementuje realizací (přerušovaná šipka s prázdným trojúhelníkem).

Nyní už se ale dostáváme k příkladu tohoto projektu. Obecně lze říci, že byl pro implementaci zvolen návrhový vzor – třívrstvý model MVC. Náš softwarový systém je totiž složen převážně ze tří typů tříd. Prvním typem je Model. Jedná se o konceptuální aplikační třídu, která reprezentuje objekty, udržuje jejich vlastnosti a chování. Pod písmenem V se ukrývá View, tedy pohled. Tyto třídy tvoří prezentační vrstvu, nebo také uživatelské rozhraní. Ve Visual Studiu jsou to typicky formuláře. Poslední částí je Controller, volně přeloženo jako ovládající třída. Objekty z této vrstvy koordinují chování aplikace, definují způsob reakce rozhraní na uživatelský vstup. Každý formulář v aplikaci má proto implicitně svůj vlastní controller. Ve zjednodušeném třídním diagramu na obrázku 21 však tyto formuláře vystupují společně.



Obr. 21 – Třídní diagram

Ve struktuře je možno vysledovat náznak ještě jednoho návrhového vzoru, kompozitu. Objekt *Zadani* se totiž skládá z několika objektů typu *Automat*. Tyto jsou unikátní pouze pro danou třídu, samostatně nemají smysl. *Zadani* tak sjednocuje deterministické i nedeterministické automaty a přidává k nim další vlastnosti. Typický příklad návrhového vzoru kompozit to ale není.

A jak bylo právě naznačeno, zde je i příklad typického použití objektově orientovaného programování. Oba typy automatů, ačkoli se svou podstatou odlišují, mají totiž několik společných vlastností, které jsou zachyceny v jejich mateřské abstraktní třídě. Stejně tak je tomu u stavů automatu. Nedeterministické se od druhých odlišují, ale rysy zůstávají stejné. Přesné složení objektů lze vysledovat přímo z diagramu.

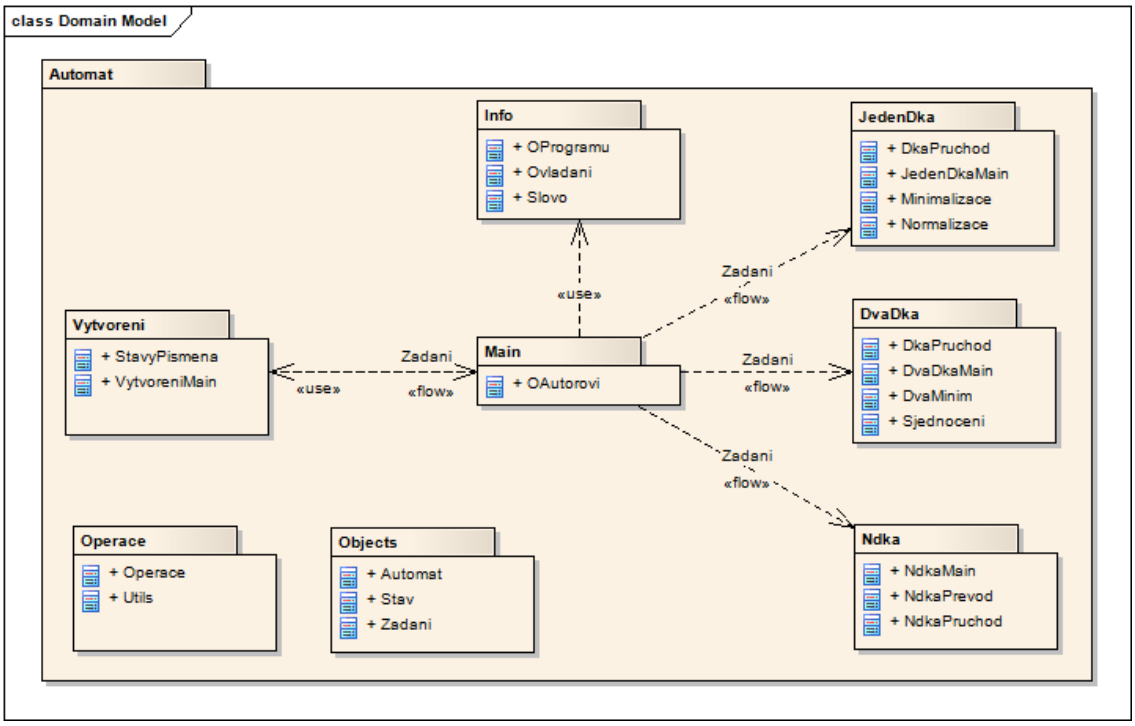
Pro názornost byl vytvořen zjednodušený třídní diagram, protože kompletní by se vlezl až na největší formát papíru A0. Obrázek 21 tedy znázorňuje základní třídy, jejich hlavní atributy a metody, a především vztahy mezi nimi. Abstrakci tříd *Automat* a *Stav* jsme již popsali, snad jen doplním, že konkrétní automat je složen z jednoho a více příslušných stavů. Třídu Zadani jsme zmiňovali v souvislosti s kompozitem, protože uchovává všechny důležité objekty a vlastnosti, které nadefinoval uživatel. Patří mezi ně až tři vytvořené automaty, zvolený jazyk i typ operací, které chceme provádět. Další trojici představují formuláře s logikou. Pro jeden deterministický i nedeterministický automat nebo pro dva DKA. Všechny tři formuláře využívají výpočetní funkce třídy Operace, které se budu věnovat podrobněji v poslední kapitole textu.

4.5.2 Struktura balíčků

Další model je velice blízký třídnímu diagramu. Popisuje rovněž statickou strukturu, tentokrát však z pohledu organizace balíčků. Názorně zachycuje rozmístění formulářů, objektů a dalších komponent a zvyšuje orientaci v projektu.

Základní diagram na nejvyšší úrovni zachycuje aplikaci Automat jako celek (obr. 22). Na první pohled je patrné, že nejdůležitější roli hraje formulář *Main*. Kolem něj se vše točí a jsou na něm napojeny ostatní třídy. Obecné informace a nápovědné texty poskytuje balíček *Info*. Hlavního menu používá vytváření automatu (*Vytvoreni*), které obsahuje formuláře *StavyPismena* a *VytvoreniMain*. V tomto balíčku vzniká objekt *Zadani*, který se pošle zpět do třídy *Main*, odkud se předává dále v rámci celé aplikace. Využije se v rámci operací nad jedním (*JedenDka*) nebo dvěma (*DvaDka*) deterministickými automaty či pro nedeterministický automat (*Ndka*). Všechny tři tyto balíčky mají své vlastní diagramy, které si podrobněji popíšeme dále v textu.

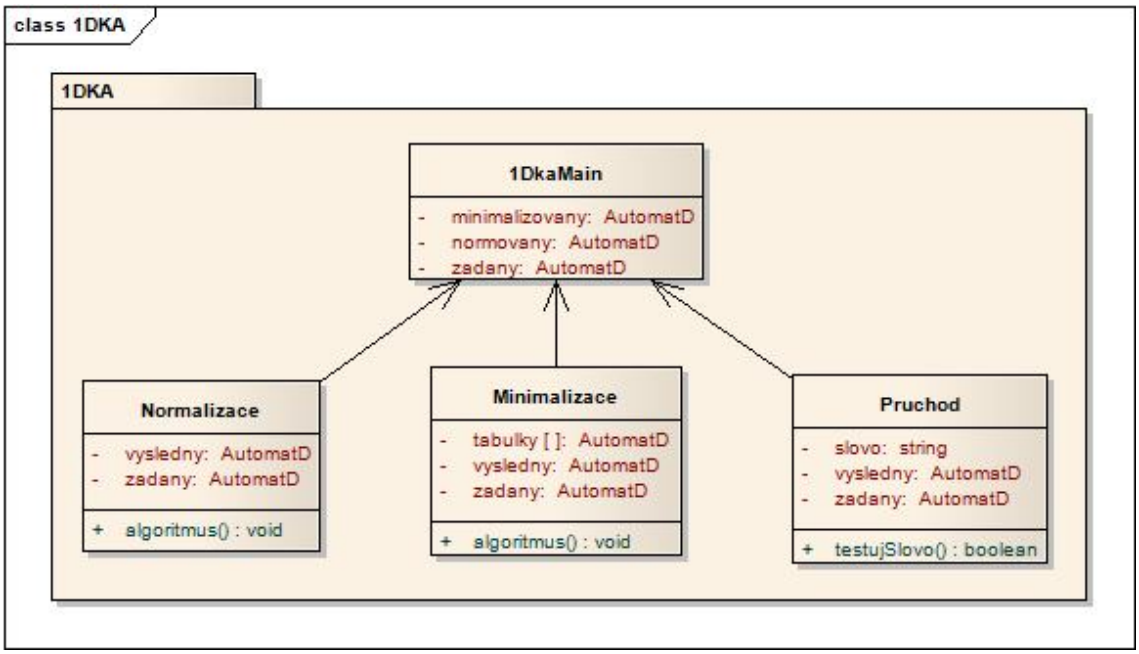
V rámci celé aplikace jsou k dispozici dva důležité balíčky *Objects* a *Operace*. V prvním jmenovaném najdeme všechny používané objekty v projektu. Nachází se zde abstraktní *Automat* i *Stav*, stejně jako jejich konkrétní potomci (*AutomatD*, *AutomatND*, resp. *StavD* a *StavND*). Také třída *Zadani* je uložena tady. Balíček *Operace* pak obsahuje třídy *Utils* a *Operace*. *Utils* poskytuje řadu užitečných a pomocných metod, většinou testovacích, třídicích apod.



Obr. 22 – Celkový model aplikace – balíčky

Další pozornost zasluhují balíčky *JedenDka*, *DvaDka* a *Ndka*. Zde se totiž odehrávají všechny hlavní operace s automaty. Postupně si je tedy představíme.

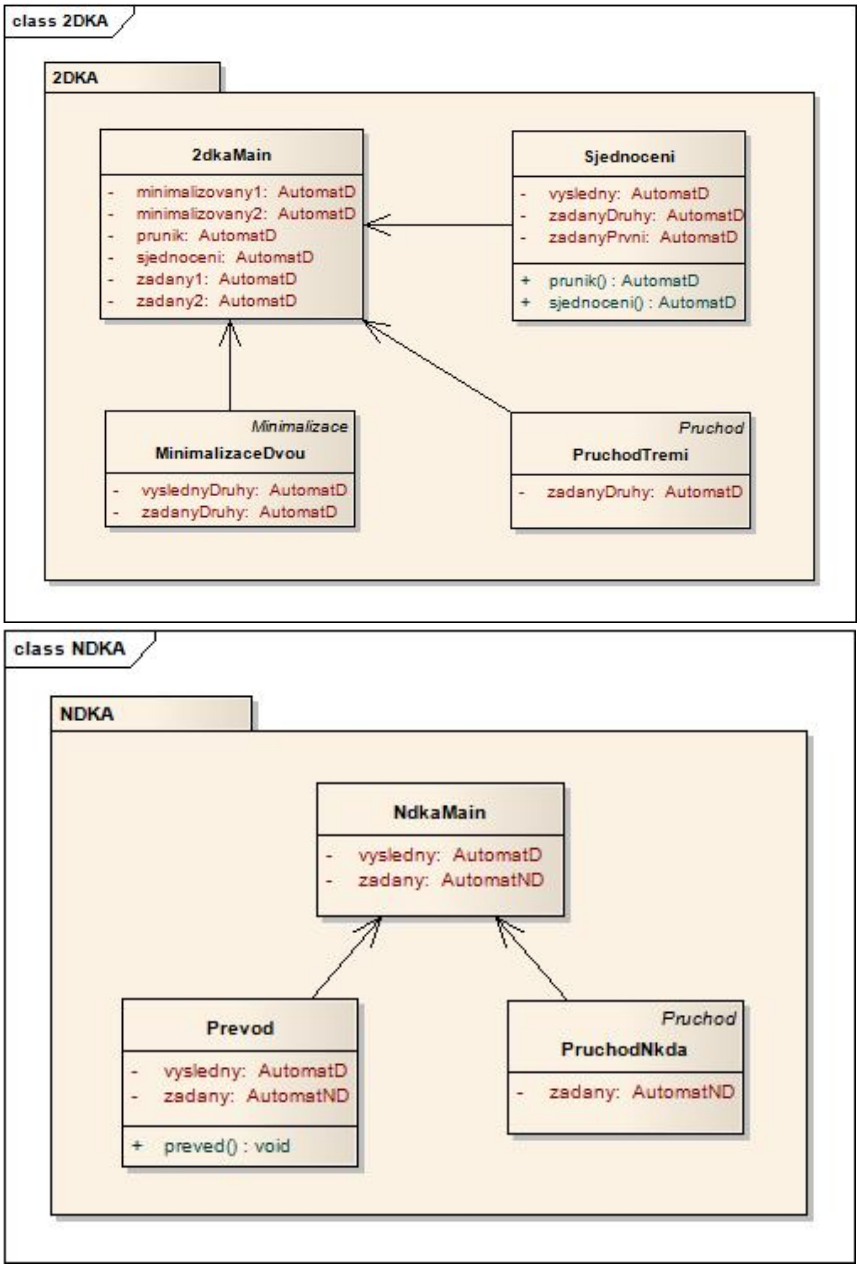
Manipulaci s jedním deterministickým automatem zajišťuje třída *1DkaMain*. Na obrázku 23 je vidět, že má pod sebou *Normalizaci*, *Minimalizaci* i *Pruchod* automatem. Uchovává také zadaný, normovaný i minimalizovaný automat ve svých atributech.



Obr. 23 – Balíček *JedenDka*

Třídy typu *Normalizace*, *Minimalizace* nebo *Prevod* (viz obr. 24 dole) mají společné rysy. Všechny potřebují na vstupu zadaný automat, který pomocí nějakého algoritmu zpracují a na výstup předají automat výsledný. Minimalizace doplňuje ještě pole přechodových tabulek.

U dvou deterministických automatů (obr. 24 nahore) musíme v hlavní třídě uchovávat informace o obou automatech. Jak jejich zadané, tak minimální verze. K tomu samozřejmě výsledné automaty po operacích sjednocení a průniku. Třída *MinimalizaceDvou* rozšiřuje základní minimalizaci, protože nepotřebuje přechodové tabulky, ale zformátovat druhý automat. Podobně je tomu také u průchodu, kde se přidává druhý zadaný graf. Třída *Sjednoceni* poskytuje rozhraní jak pro sjednocení dvou automatů, tak i pro průnik. Algoritmicky se totiž tyto dvě operace liší jen ve vyhodnocování přijímajících stavů.



Obr. 24 – Balíčky *DvaDka* (nahoře) a *Ndka* (dole)

Posledním balíčkem je *Ndka* (obr. 24 dole), který má kontrolu nad operacemi s jedním nedeterministickým automatem. Uchovává jak zadaný, tak už deterministický výsledný automat, a poskytuje rozbor převodu jednoho automatu na druhý. Třída *PruchodNdka* pak rozšiřuje základní průchod, protože zadaný automat při procházení umožňuje nacházet se současně ve více stavech.

Tolik ke statické struktuře aplikace Automat. Tím uzavíráme fázi implementace. Vysvětlovat samotný programový kód by zabralo mnoho prostoru. Zájemce ale odkážu na zdrojové soubory a projektovou dokumentaci. Každá třída je totiž popsána v hlavičce způsobem, který umožňuje vygenerovat dokumentaci ve formátu xml. Stejně tak jsou samozřejmě popsány všechny atributy i metody. V tělech procedur a funkcí pak najdete podrobné komentáře k daným blokům i jednotlivým částem zdrojového textu.

4.6 Testování a předání

Testování probíhá z pohledu tří základních dimenzí reprezentovaných kvalitou, funkcionalitou a výkonností systému. Týká se všech vytvářených modelů a jejich diagramů. Pro účelnost testování je klíčové, aby ověřování kvality a funkčnosti probíhalo průběžně. Všechny diagramy z fází specifikace požadavků, analýzy i návrhu byly proto konzultovány s vedoucím diplomové práce již v předstihu, aby nedošlo při vývoji k závažnějším chybám.

Spustitelné programy, výsledky jednotlivých iterací, rovněž putovaly k přetestování. Nejdříve vznikla beta verze, kdy byla představena základní kostra systému s možností vytvořit a nadefinovat automat. Byly odhaleny chyby jak v uživatelském rozhraní, tak ve výkonném kódu. Vzniklo také několik doplňujících či upřesňujících požadavků, které byly spravovány a průběžně komponovány do programu.

Samotné testování probíhalo tedy na dvou úrovních. Nejprve jsem já z pozice vývojáře otestoval funkcionalitu i správnost algoritmů. Mohl jsem k tomu využít krokovací nástroje, které prostředí Visual Studia nabízí ve velmi vysoké kvalitě. Odladěnou verzi jsem pak zaslal na přezkoumání vedoucímu DP, který otestoval program z jiného pohledu. Společně jsme tak odhalili několik nedostatků, dokonce i chyby v důležitých algoritmech.

Stabilitu aplikace zajišťuje správa výjimek, prakticky se tak nemůže stát, že by došlo k jejímu pádu. Veškeré vstupy od uživatele procházejí typovou kontrolou a nevhodné údaje jsou odmítnuty. Ukládání a načítání souboru na disk podléhá rovněž přísným pravidlům. Uživateli tedy nebyly ponechány žádné prostředky pro shození aplikace, která se tak stala téměř neprůstřelnou.

Správnost algoritmů byla ověřena opět důkladně. Na typových příkladech se kontrolovaly výsledné automaty, navíc lze průběh operací sledovat po krocích přímo v aplikaci, takže prostor pro možný výskyt chyb se značně zužil. Mimo jiné byla také použita tzv. metoda pokus-omyl, která prověřila všechny stránky programu asi nejvíce. Náhodně vytvářené vstupy často generují podivuhodné a nečekané výsledky, které je potom nutné podrobit detailnějšímu zkoumání. Mnoho chyb se také odhalilo pomocí průchodu zadaných a výsledných automatů, kdy slovo musí být přijímáno výslednými automaty stejně, jako těmi vstupními.

Jak již bylo řečeno, produkt se předával po každé větší ukončené iteraci. Tím vznikla zpětná vazba mezi vývojářem a zadavatelem. Nemohlo se tedy stát, že bude odevzdána diplomová práce, která nesplňuje zadání. Navíc bylo možno přizpůsobit aplikaci požadavkům a potřebám vedoucího DP, což prospělo kvalitě produktu a tím i oběma stranám. Diplomová práce tak splňuje všechny body zadání, které navíc v některých ohledech mírně přesahuje. Má však i své rezervy, především v uživatelské přívětivosti. S těmi byl ale zadavatel seznámen ještě před odevzdáním finální verze, právě při testování.

5 Třída Operace

Třída *Operace* představuje hlavní výkonné jádro systému. Právě zde jsou uloženy veškeré algoritmy, se kterými se setkáváme napříč aplikací. V následujících kapitolách si představíme hlavní metody, jejichž těla popíšeme přirozeným jazykem.

Než se k nim ale dostaneme, je nutné zmínit také třídu *Utils*, kterou *Operace* v hojné míře využívá. Třída *Utils* obsahuje především podpůrné algoritmy a výpočty, které mohou být použity jak v aplikační logice, tak ke kontrole vstupů na vrstvě prezentační. Počítají se zde například pozice komponent pro konkrétní počet stavů, vyhodnocuje se platnost přechodů nebo se manipuluje s polem stavů, které se dle požadavku rozšiřuje a zmenšuje. Užitečný je určitě i algoritmus seřazení čísel stavů.

5.1 Normování

Základní operace s jedním deterministickým automatem je jeho převod do normovaného tvaru. Definici tvaru najdete v kapitole 3.3, nás ale zajímá, jak to vyřešit algoritmicky.

Vstup: zadaný deterministický automat (objekt typu *AutomatD*)
Výstup: převedený deterministický automat (*AutomatD*)
Postup:

1. Projdu celý automat a na první pozici výsledného uložím počáteční stav
2. Postupně pro všechny stavy výsledného automatu (který se po každém průchod zvětší)
   a) Načtu čísla stavů (přechody) přes písmena A, B a v případě větší abecedy i C
   b) Pro všechny tyto přechody provedu kontrolu, zdali jsou už obsaženy ve výsledném automatu.
   c) Pokud ano, přeskočím na další přechod. Pokud ne, do výsledného automatu přidám stav s číslem příslušného přechodu.
   d) Pokračuji dalším přechodem. Nezbývá-li žádný, přesouvám se na další stav ve výsledném automatu.
3. Prošli jsme všechny dosažitelné, ale i nedosažitelné stavy.
4. Nedosažitelné stavy je třeba odstranit.
5. Výsledný automat má uloženy stavy ve správném pořadí, ale stavy i přechody jsou špatně očíslovány.
6. Přečíslujeme stavy i přechody.
7. Vrátíme výsledný automat v normovaném tvaru.

5.2 Minimalizace

Nejsložitějším algoritmem v aplikaci je bezesporu minimalizace. Byla několikrát přepisována pro požadavky krokování, stejně tak z důvodu chybovosti pro některé vstupy. A takto vypadá finální verze.

Vstup: zadaný deterministický automat (*AutomatD*)
Výstup: pole deterministických automatů, kde poslední je výsledný minimalizovaný
Postup:

1. Do připraveného pole automatů uložíme na první pozici zadaný automat
2. Na pozici druhou ten samý automat, kde jsou ale nepřijímající stavy zařazeny do množiny 2 a přijímající do množiny 1.
3. Pro každý stav automatu z poslední pozice v poli uložíme čtveřici: přechod přes A, B a C a číslo koncové množiny.
4. Tyto čtveřice z minulého kroku uchovávají nové přechody do množin, následuje rozdělení a přečíslování nových množin.
5. Automat s novými přechody a přečíslovanými množinami uložíme do výsledného pole.
6. Pokud je číslo nejvyšší množiny menší, než počet stavů, pokračujeme znovu bodem 3.
7. Jinak je minimalizace u konce. Převedeme poslední automat do normovaného tvaru.
8. Uložíme automat na poslední pozici do výsledného pole, které pošleme na výstup.

5.3 Převod NDKA na DKA

Zajímavý je také převod nedeterministického automatu na deterministický. Když se nad ním zamyslíme, zjistíme, že algoritmus je velice podobný normování. Jen se zde nepracuje s jednotlivými čísly stavů, ale s celými množinami přechodů.

Vstup: zadaný nedeterministický automat (*AutomaNtD*)
Výstup: převedený deterministický automat (*AutomatD*)
Postup:

1. Projdu celý automat a na první pozici výsledného deterministického automatu uložím množinu všech počátečních stavů. Jen místo deterministických stavů zatím používáme nedeterministické.
2. Postupně pro všechny množiny stavů výsledného automatu (kterých po každém průchodu přibude)
    a) Načtu množiny stavů (přechodů) přes písmena A, B a v případě větší abecedy i C
    b) Pro všechny tyto přechody provedu kontrolu, zdali jsou už obsaženy ve výsledném automatu.
    c) Pokud ano, přeskočím na další přechod. Pokud ne, do výsledného automatu přidám množinu stavů podle příslušného přechodu.
    d) Pokračuji další množinou přechodů. Nezbývá-li žádná, přesouvám se na další stav ve výsledném automatu.

3. Prošli jsme všechny dosažitelné, ale i nedosažitelné množiny stavů.
4. Nedosažitelné stavy je třeba odstranit.
5. Výsledný automat má uloženy množiny stavů ve správném pořadí, tyto množiny i přechody je však nutné přejmenovat a přečíslovat.
6. Přečíslujeme stavy i přechody, čímž získáme výsledný automat.
7. Vrátíme výsledný deterministický automat v normovaném tvaru.

5.4 Sjednocení a průnik

Posledním sofistikovanějším algoritmem je operace sjednocení. Záměrně nezmiňuji průnik, protože ten je součástí algoritmu automaticky a pomocí indikátoru se provádí větvení přímo uvnitř kódu.

Sjednocení dvou deterministických automatů však vykazuje známky dvou předchozích postupů. Jedním je normování a druhým převod NDKA na DKA. Obzvláště převod je velmi blízký, protože se v každém kroku skládají dva přechody, tedy pokaždé se ze dvou výchozích stavů dostaneme do jiných dvou (klidně i těch samých). Normované pořadí je pak zaručeno konstrukcí výsledného deterministického automatu, který však v průběhu procesu musí obsahovat nedeterministické stavy.

Vstup: dva zadané deterministické automaty (*AutomatD*)
Výstup: výsledný deterministický automat (*AutomatD*)
Postup:

1. Projdeme první i druhý zadaný automat a najdeme počáteční stavy.
2. Spojíme tyto stavy do jednoho, spojíme i přechody a uložíme tento nedeterministický stav, přičemž dáváme pozor na pořadí, které nikdy nesmíme zaměnit! (například setříděním)
3. Vyhodnotíme, zdali bude tento stav přijímající – pokud byl aspoň jeden ze stavů, pak je i výsledný (v případě průniku musejí být oba)
4. Průběžně udržuje seznam a počet přidaných stavů (v této době jeden počáteční) a pozici v těchto stavech (nyní první)
5. Pokud je počet stavů menší nebo roven aktuální pozici (stavu), pak
   a) Ulož čísla přechodů z obou automatů zvlášť pro písmena A, B a C (tím vznikne dvojice, která jednoznačně určuje stav)
   b) Pro každou získanou dvojici otestuj, je-li už v seznamu přidaných stavů
   c) Pokud ano, přeskoč na další dvojici
   d) Pokud není, přidej dvojici do seznamu přidaných stavů, vyhodnoť přijímající stav a pokračuj další dvojicí
   e) Jakmile dojdou dvojice, pokračuj bodem 5.
6. V tomto bodě máme uloženy všechny nedeterministické stavy výsledného automatu v normovaném pořadí
7. Zbývá je přejmenovat, přečíslovat a výsledný deterministický automat poslat na výstup.

Třída Operace obsahuje i další metody, většina z nich ale dále upravuje výše představené algoritmy a zpracovává jejich výsledky. To je potřeba hlavně při procházení a vysvětlování postupů na formulářích. Okno s uživatelským rozhraním už bude mít k dispozici přesně ta data, která bez úprav zobrazí v přechodových tabulkách. Krokování pak bude zajištěno postupným odkrýváním nebo zvýrazňováním aktivních položek či celých komponent.

V této třídě by asi mnoho z vás očekávalo průchod automatem na základě zadaného slova, je třeba si ale uvědomit, že na tom není co algoritmicky řešit. Stačí převzít slovo a k němu vytvořit posloupnost aktivních stavů pro dané symboly. Jediný rozdíl bude u nedeterministického automatu, kde místo jednotlivých stavů budou celé množiny.

Během celého vývoje se ukázalo velice výhodné oddělit tyto složité bloky kódu od zbytku aplikace. V případě nalezených chyb nyní stačí jen vyměnit výkonnou část a zbytek programu zůstane nedotčen. Zásahy do kódu lze tedy provádět, aniž bychom ohrozili funkčnost systému jako celku. Nic přitom nebrání dalšímu dynamickému rozvoji aplikace někdy v budoucnu.

6 Závěr

Svou diplomovou práci jsem nasměroval cestou za poznáním a získáváním vědomostí, abych pomohl dosáhnout magisterského stupně vzdělání nejen jednorázově sobě samému, ale také v budoucnu všem případným zájemcům o navazující studium na fakultě elektrotechniky a informatiky Vysoké školy Báňské v Ostravě.

Výsledkem mé snahy je celkem robustní aplikace, která byla navržena a vyvíjena moderním softwarovým procesem, a proto nebude těžké ji v budoucnu znovu rozšířit. Také po stránce uživatelského prostředí a grafického rozhraní se podařilo skloubit jednoduchost s intuitivností, prakticky celý program lze ovládat pouze myší. K lepší orientaci poslouží propracované menu, kde bych chtěl vyzvednout možnost přepnutí do anglického jazyka.

Samotná funkcionalita je založena především na univerzálnosti a znovu použitelnosti. Pod pojmem prvním se skrývá variabilita zadávání automatů, kde takřka nejste omezeni. Vybírat lze mezi deterministickým i nedeterministickým konečným automatem, počet stavů v běžných příkladech málokdy překročí číslo dvanáct a samotné přechody nebo přijímající stavy je možné nadefinovat naprosto dle Vašich představ. A význam slov „znovu použitelnost“ asi nemusím dlouze vysvětlovat. Program totiž umožňuje všechny zadávané i výsledné automaty uložit do souboru, a proto není problém se k nim v budoucnu kdykoli vrátit.

A aby se mohl jakýkoli program stát využívanou učební pomůckou, musí být ve vysvětlování názorný. Rovněž tuto vlastnost jsem se snažil do projektu a hlavně všech algoritmů zakomponovat. Postupy jsou jednoznačné, účelné a detailně krokovatelné, přičemž každý krok je dostatečně popsán nápovědným textem.

Pevně věřím, že výuková aplikace pro předmět teoretická informatika najde mezi studenty své uplatnění. Má k tomu i dobré předpoklady. Na internetu nejsou totiž aplikace podobného typu volně ke stažení, a když už na nějakou takovou narazíte, jedná se z převážné většiny o komerční produkt. Vlastníkem mé aplikace, jakožto i celé diplomové práce, je však naštěstí naše univerzita, která o své studenty vždy pečovala dobře a zajišťovala jim přístup k novým učebním pomůckám i moderním metodikám.

Použitá literatura

1. Petr Jančar, Teoretická informatika, výukový text FEI VŠB (2004)
http://www.cs.vsb.cz/jancar/TEORET-INF/teoret-inf.htm
4.5.2009

2. Doc. RNDr. Petr Hliněný, Ph.D., Úvod do teoretické informatiky (2005)
výukový text, verze 0.99

3. Wikipedie České Republiky, otevřená encyklopedie
http://cs.wikipedia.org/wiki/Kone%C4%8Dn%C3%BD_automat
4.5.2009

4. Hubert Dostál, Teorie konečných automatů, regulárních gramatik, jazyků a výrazů (2008)
http://iris.uhk.cz/tein/teorie/konecnyAutomat.html
4.5.2009

5. Rational Unified Process
http://www.ts.mah.se/RUP/RationalUnifiedProcess/process/artifact/ovu_arts.htm
4.5.2009

6. Scott W. Ambler, Agile Modeling and the Rational Unified Process (RUP)
http://www.agilemodeling.com/essays/agileModelingRUP.htm
4.5.2009

7. Petr Puš, Poznáváme C# a Microsoft .NET – 1.díl
http://www.zive.cz/h/Programovani/AR.asp?ARI=119978
4.5.2009

8. Visual C# Developer Center
http://msdn.microsoft.com/en-us/vcsharp/default.aspx
4.5.2009

Příloha I – Obsah CD

AUTOMAT\	– spustitelná aplikace
C#\	– zdrojové kódy (včetně historie vývoje)
DOKUMENTACE\	– veškeré dokumenty (UML, texty)
FRAMEWORK\	– instalační soubor Microsoft .NET Framework
PRIKLADY\	– uložené příklady automatů
ZALOHA\	– záloha důležitých části DP (archiv ZIP a RAR)
ctime.txt	– abstrakt a klíčová slova v češtině
Diplomova_prace.pdf	– text diplomové práce
readme.txt	– abstrakt a klíčová slova v angličtině

Příloha II – Uživatelská dokumentace